UNIFICATION OF LARCH AND Z-BASED OBJECT MODELS TO SUPPORT ALGEBRAICALLY-BASED DESIGN REFINEMENT: THE LARCH PERSPECTIVE

THESIS

Catherine J. Lin
Captain, USAF

AFIT/GCS/ENG/94D-15

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

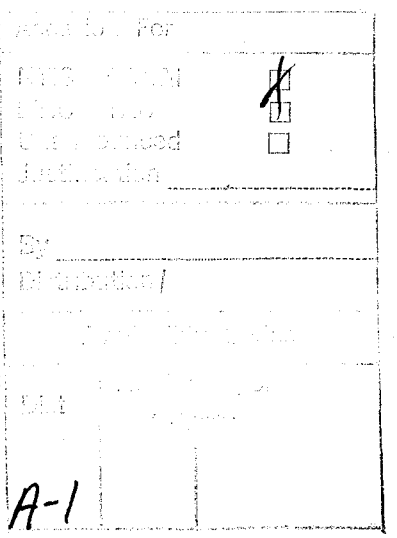
Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/94D-15

UNIFICATION OF LARCH AND *Z*-BASED OBJECT MODELS TO SUPPORT ALGEBRAICALLY-BASED DESIGN REFINEMENT: THE LARCH PERSPECTIVE

THESIS
Catherine J. Lin
Captain, USAF

AFIT/GCS/ENG/94D-15



Approved for public release; distribution unlimited

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1994	Master's Thesis

4. TITLE AND SUBTITLE
UNIFICATION OF LARCH AND Z-BASED OBJECT MODELS TO SUPPORT ALGEBRAICALLY-BASED DESIGN REFINEMENT: THE LARCH PERSPECTIVE

5. FUNDING NUMBERS

6. AUTHOR(S)
Catherine J. Lin, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/94D-15

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
NSA, 9800 Savage Road, Fort Meade, MD, 20755-6000

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
This research describes the feasibility of developing object-oriented LARCH specifications, part of a dual approach for formally extending object-oriented analysis models using LARCH and *Z*. The first phase consisted of two steps: establishing a set of transformation heuristics for algebraically representing object models and implementing a robust LARCH parser. The LARCH parser produced abstract syntax trees (ASTs) of objects forming the basis for analyzing the similarities and differences between *Z*-based and LARCH-based object representations. The second phase used the analysis of LARCH and *Z* to identify fundamental core constructs in the languages and abstract syntax trees. These core constructs consisted of similar syntactic and semantic notions of signatures and axioms for describing a problem domain, thereby forming a canonical framework for formal object representations. This canonical framework provides a front-end for producing design refinement artifacts such as interface languages, theorem proving sentences, and synthesis diagrams. The final phase demonstrated the feasibility of interface language generation by establishing an executable framework. This executable framework mapped LARCH into the SOFTWARE REFINERY^{TM} Programming Environment to rapidly prototype object-oriented LARCH specifications.

14. SUBJECT TERMS
computers, computer programs, software engineering, specifications, formal specification languages, application composition systems, Larch

15. NUMBER OF PAGES
165

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to *stay within the lines* to meet *optical scanning requirements*.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 January 1988). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	PR	- Project
G	- Grant	TA	- Task
PE	- Program Element	WU	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es) Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD	- See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE	- See authorities.
NASA	- See Handbook NHB 2200.2.
NTIS	- Leave blank.

Block 12b. Distribution Code.

DOD	- Leave blank.
DOE	- Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA	- Leave blank.
NTIS	- Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

AFIT/GCS/ENG/94D-15

Unification of Larch and *Z*-Based Object Models To Support Algebraically-Based Design Refinement: The LARCH Perspective

THESIS

Presented to the Faculty of the Graduate School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science

Catherine J. Lin, B.S.E.E. and B.S.E.P.P.
Captain, USAF

December 13, 1994

Acknowledgements

I would like to thank my advisor, Major Paul Bailor, for his guidance and assistance during this research effort. I also wish to thank Major David Luginbuhl for his support and help in policing those "dangling" modifiers. Thanks also to Dr. Potoczny for being on my committee.

I am grateful to the other members of the Knowledge Based Software Engineering (KBSE) research group for all their support, humor, and kindness. They offered me timely diversions to recalibrate my perspective here at AFIT. I am especially thankful to Kathleen Wabiszewski, "KK", for being a great research partner and providing a good sounding board for listening to my ideas and concerns. Additionally, I would like to thank Dr. Perry Alexander and Major Scott Deloach for providing me insightful comments and suggestions for my research efforts.

Finally, I would like to thank my husband, Jeff, for all his support, understanding, and most of all confidence in me. I appreciate the sacrifice he made for coming to Dayton, and the hours of laughter he has given me by taking a lighter look on life.

Catherine J. Lin

Table of Contents

List of Figures

List of Tables

Unification of Larch and *Z*-Based Object Models To Support Algebraically-Based Design Refinement: The LARCH Perspective

I. Introduction

1.1 Background

The software development community has progressed from producing batch-oriented, single application programs to delivering highly distributed, parallel applications. As software systems grow in sophistication, capturing software requirements with precision becomes imperative. In spite of this requirement, most specifications are written using natural languages such as English text and graphical notations. These specifications are ambiguous and imprecise leaving them open to interpretation (NS91). In addition, informal specifications exacerbate the development process by causing software designers to miss requirements and by increasing the risk of delivering an undesired product.

Therefore, in an attempt to introduce rigor into the specification process, software researchers are developing and applying formal specification techniques. Formal specification languages are precise and unambiguous because they are based on fundamental mathematical constructs such as set theory and predicate logic (Win87). These languages provide software engineers with the capability to express complex ideas in a concise fashion (NS91). In addition, software engineers can reason about their specifications and form

proofs of correctness (NS91). Thus, this theoretical basis provides the clarity and precision required for specifying software systems (Sri91).

Although formal specification languages provide a powerful mechanism for analyzing and defining a software problem, many software developers are leery of the approach. They claim that formal techniques require a sophisticated mathematics background and take too long to learn (Hal90). More importantly, they find it difficult to communicate the mathematical description to a system's user. Consequently, many software developers rely on a less rigorous object-oriented approach during the specification phase.

In the last decade, object-oriented methods have emerged as a practical way to organize information about a problem (RBP+91). Developers use object-oriented models to analyze a system by decomposing it into smaller, more manageable modules. In addition, object-oriented models help a system's user visualize his system. Therefore, the use of these models has become one of the preferred methods for documenting software requirements.

To exploit the strengths of both the practical object-oriented approach and the more precise theory-based perspective, the software engineering community is exploring ways to formally extend the object-oriented analysis process. A proposed approach is being developed at the Air Force Institute of Technology (AFIT) by the Knowledge-Based Software Engineering (KBSE) research group. To date, the group has developed a domain-specific application composition system that relies on an object-oriented approach for documenting domain knowledge. This domain knowledge is translated into an object-based environment known as SOFTWARE REFINERY^{TM} using REFINE, a wide-spectrum, Lisp-based specification language. Once the domain objects have been defined, Architect, an appli-

cation composition system based on a formalized software architecture, composes domain specific applications with its toolkit of components, connectors, and constraints. These applications can be executed simulating a specification's behavior and allowing specifiers to evaluate whether the demonstrated behavior matches the desired behavior.

The advances achieved in the KBSE object-based work have led to next-generation research in integrating formal methods with object-oriented analysis and design techniques. Figure 1.1 shows two complementary efforts that explore the feasibility of transforming object-oriented models into formal representations. The right path employs an algebraic-based approach for specifying object-oriented models while the left path establishes a *Z*-based framework. Coupled with these two object transformations is a unification mechanism that produces a unified abstract framework. The basis for a unified model is to reduce algebraic and *Z* specifications into a canonical representation establishing a general model for formalized object models and promoting the notion of class-based solutions. Two paths are possible from the unification phase, a design refinement phase and an executability analysis phase. The design refinement phase refines an abstract specification into a concrete description providing an adequate interface into the design process. The executability analysis phase establishes an execution framework for a specification allowing a specification to be demonstrated and validated.

1.2 Problem

This research studies the algebraic-based perspective, the right path in Figure 1.1, focusing on the following objective:

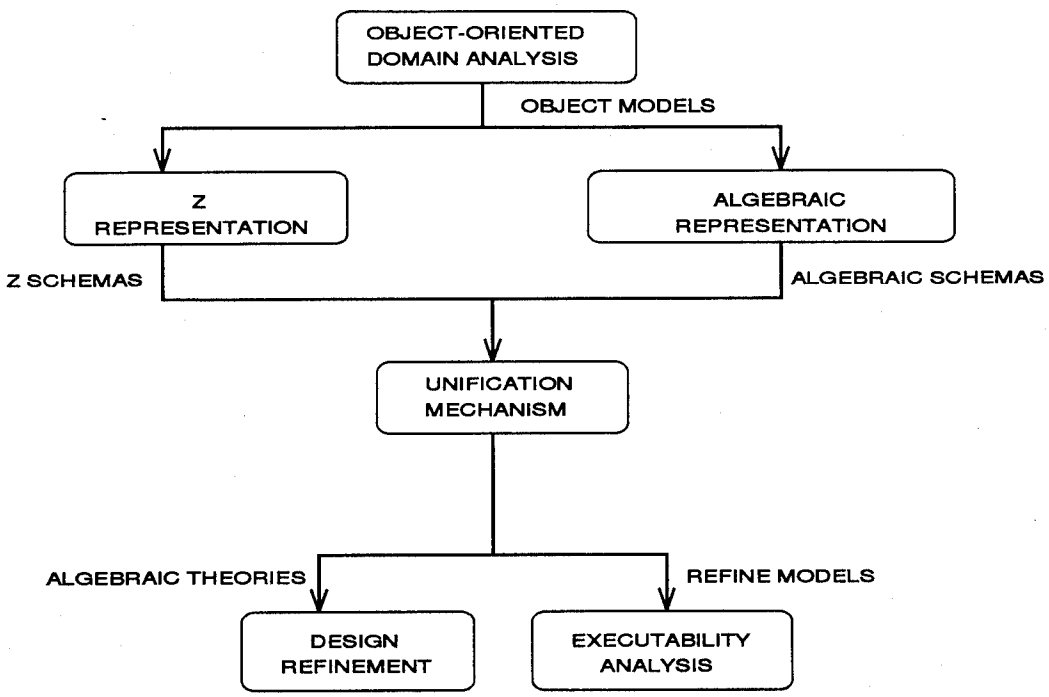


Figure 1.1 Formalized Object Transformations

Formally extend object-oriented analysis models using algebraic specifications to create an executable framework, while incorporating a unification with Z-based models.

Accomplishing this objective requires a three-step process. First, an algebraic framework for object-oriented analysis models needs to be established. While this research addresses formalizing object models using algebras, Wabiszewski is expanding the *Z*-based perspective (Wab94). These paths are linked by a cooperative effort in creating a unifying abstract framework. Therefore, the second step focuses on analyzing the similarities and differences between algebraic and *Z* specifications to design a unified model. In order to scope the remainder of this study, the final step concentrates on creating a stable execution framework based on the unified model. The design refinement phase is left for future research.

1.3 Research Approach

For this study, Rumbaugh's Object Modeling Technique (OMT) (RBP+91) was selected as the object-oriented approach because of the researcher's familiarity with the technique as well as concurrent work that uses the Rumbaugh method (Wab94). To meet the research objective described in Section 1.2, the following approach has been established:

1. Design and develop an algebraic transformation process using object-oriented analysis (OOA) models as the source representation and algebraic specifications as the target representation.
   (a) Select an appropriate algebraic language for the target framework.
   (b) Establish an evolutionary approach for the design and development of the transformation process.
   (c) Identify and document the strengths and weaknesses in the transformation process.
2. Analyze the complementary paths shown in Figure 1.2 and develop a unifying abstract structure.
3. Design an initial execution framework based on the unified model.

1.4 Sequence of Presentation

The remainder of this thesis is organized as follows:

Chapter II reviews Rumbaugh's Object Modeling Technique and the current literature in the field of algebraic specifications. This literature review also presents several approaches for formalizing object-oriented techniques.

Chapter III describes an evolutionary development approach for the formal object transformation process, presents the algebraic transformation phase, and introduces the formal execution phase. In addition, the validation domains for the transformation process are described.

Chapter IV contains a design and implementation of an algebraic language parser.

Chapter V describes the motivation behind establishing a unified abstract model and presents a design and implementation of the Unified Model. Additionally, this chapter describes the validation process for the unified abstract framework.

Chapter VI presents an initial design and implementation of a formal execution framework for LARCH and *Z*.

Chapter VII contains general conclusions and future recommendations for research in this area.

Several appendices are included to provide additional information.

II. Literature Review

2.1 Introduction

To support the research goals outlined in Chapter I, this chapter reviews two relevant topics: Rumbaugh's (RBP+91) object-oriented analysis method and the origin and evolution of algebraic specifications. Rumbaugh's Object Modeling Technique (OMT) is the source for the formal object transformation process, and Section 2.2 describes the OMT analysis models and their key object-oriented characteristics. After establishing an understanding of the source model, Section 2.3 examines the feasibility of formally representing OMT models using algebraic specifications. Finally, Section 2.4 evaluates three approaches for formalizing the object-oriented methodology.

2.2 The Rumbaugh Object Modeling Technique (OMT)

OMT is an object-oriented methodology that focuses on software analysis using three models to scope an application domain. These models represent the structural organization of the domain, the reactive behavior of the domain, and the data transformations of the domain. The models are named the object, dynamic, and functional models, respectively. The object model is the keystone for the OMT analysis process and provides a foundation from which to construct the dynamic and functional models. Together these models form the framework for organizing and representing domain information.

2.2.1 OMT Object Models. The object model captures the structure of a software domain by identifying the important objects, attributes, and associations between objects (RBP+91). Two important associations in the object model are inheritance and

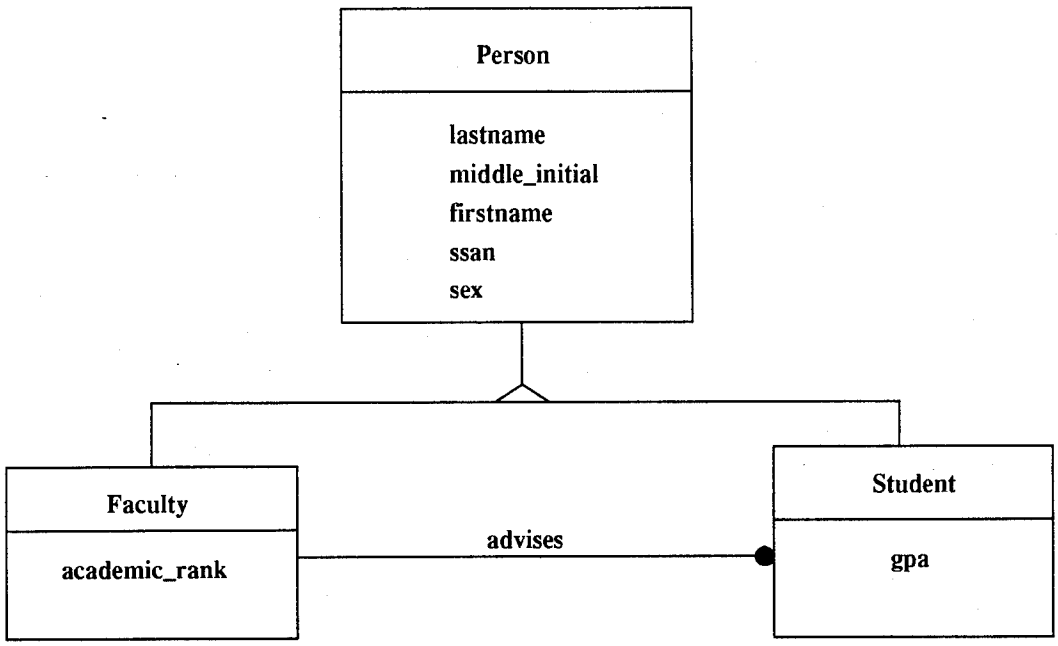


Figure 2.1 Faculty and Student Object Model

aggregation. Inheritance establishes an "is-a" relationship between a general object and its specialization. Aggregation identifies the "is-composed-of" relationship associating an assembly with its individual components (RBP+91). Based on these associations, the object model communicates the organization and important dependencies of a software system to the software developer. At the end of the object modeling phase, an object model diagram and associated data dictionary are produced. As an example, an object model depicting an inheritance relationship and association between the person, faculty, and student objects is shown in Figure 2.1 (HB94).

2.2.2 OMT Dynamic Models. The OMT dynamic model uses two fundamental modeling abstractions: events and states to model an object's overall state space and subspaces. An event represents external stimuli that causes an object to behave in a certain manner. An object's state represents the current values of an object's attributes

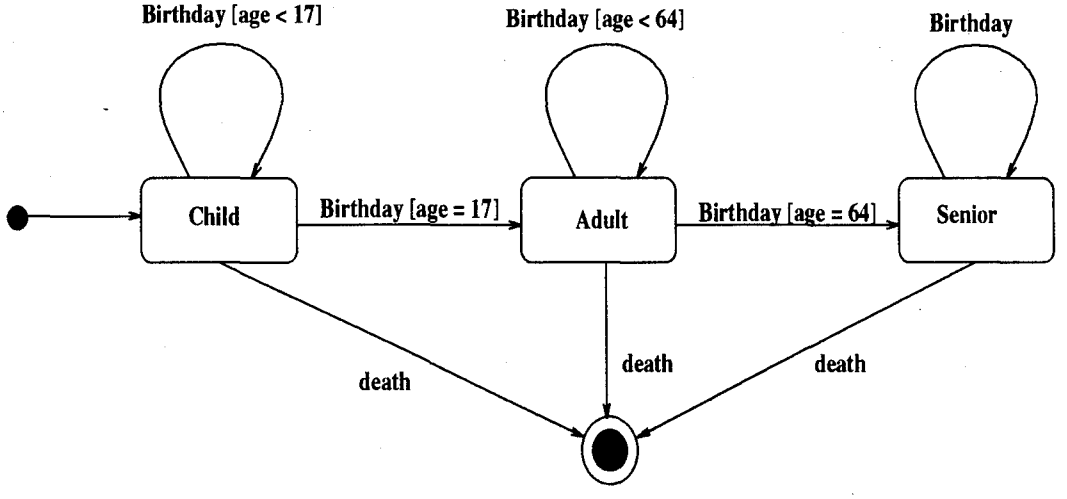


Figure 2.2 Person's Age Dynamic Model

after an event has occurred. Consequently, an object may have several states based on the number of events and corresponding state transitions. In the end, this phase generates two products: a set of dynamic models representing a domain's state space and several corresponding state transition tables describing the sequence of events. In relation to the person object model described in the previous section, Figure 2.2 (HB94) shows a dynamic model for describing the various states of a person as a person ages, i.e., child, adult, and senior. Table 2.1 illustrates a simplified state transition table.

Table 2.1 Person State Transition Table.

Current State	Event	Parameters	Guard	Next State	Action
Child	Birthday		age < 17	Child	
Child	Birthday		age = 17	Adult	
Child	death			Terminated	
Adult	Birthday		age < 64	Adult	
Adult	Birthday		age = 64	Senior	
Adult	death			Terminated	
Senior	Birthday			Senior	
Senior	death			Terminated	

2.2.3 OMT Functional Models. Rumbaugh uses the functional model to describe the operations defined in the object model and the actions defined in the dynamic model (RBP+91). Data transformation is the functional model's central concept; the transformation is used to compute an object's output values from input values to determine an object's state. In addition, the functional model identifies all of the key consumers and producers of an object's data. The products from this phase consist of a set of data flow diagrams. Completing the person example, Figure 2.3 (HB94) shows a functional model computing the age attribute for a person.

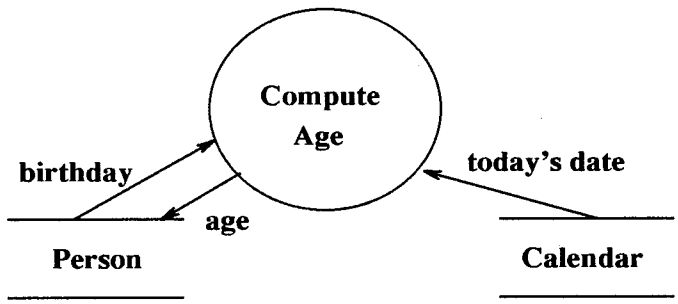


Figure 2.3 Age Data Flow Model

2.2.4 OMT Summary. The object, dynamic, and functional models represent the OMT products developed during software analysis. Depending on the type of system to be modeled, some of the three model types are not necessary. For example, database systems store and retrieve data, but these systems do not have extensive functional models since data transformations do not take place (RBP+91). On the other hand, compilers translate one language into another language and have intensive data transformation systems requiring extensive functional models (RBP+91). As software developers begin to

understand each OMT model's purpose and relationship to the other models, the OMT process assists them in documenting their software requirements more completely.

2.3 Algebraic Specifications

Algebraic specifications are a formal technique using equational logic as its syntactic domain and universal algebra and category theory for its semantic domain (EMCO92). The algebraic notation consists of a signature that defines the sorts and operators and a set of axiomatic equations that specify an operator's behavior. Originally intended for describing data types in an implementation independent manner, algebraic specifications have evolved to encompass the entire specification phase (EMCO92). The following subsection describes this evolution.

2.3.1 Using Algebraic Theories to Specify Data Types, Algorithms, and Architectures.

Formal methods researchers first used algebraic specification techniques to define the behavior of abstract data types because they believed that data types could be modeled as many-sorted algebras (EMO92). Thus, they used algebraic specifications with two goals in mind:

- structure abstract data types (ADTs) as a notion of classes
- hide the implementation dependent details by focusing on the ADT's behavior and abstract data structure

These goals established the importance of specifying data types algebraically, improving the understanding and precision of abstract data types and promoting software reuse.

spec Sorting =
based on
    Bag, Sequence, Order
operations
    sort: Bag $\rightarrow$ Sequence
axioms
    sort(b) = s if b = seq-to-bag(s)
                $\wedge$ x precedes y in s $\Rightarrow$ x $\leq$ y

Figure 2.4 A Specification for Sorting

As researchers gained experience in formally modeling ADTs, they began investigating the feasibility of formally specifying algorithms. Researchers realized that algebraic theories were well-suited for describing the fundamental properties of algorithms (Sri91) (Smi90). Srinivas demonstrated this by formally modeling the sort algorithm (Sri91), as depicted in Figure 2.4 (Sri91). He based the sort theory on the following properties: a bag, a sequence, and an order. After specifying the essential properties, Srinivas defined a sort function using a sort operator with a bag of random elements as its domain and a sequence of elements as its co-domain. To complete this specification, he used an axiomatic equation to preserve the property of the sequence.

The abstract sort specification in Figure 2.4 demonstrates the concise and exact nature of describing algorithms algebraically. This specification can be checked for consistency by deriving a set of algebraic equations from the specification's axiom. These equations are derived under the closure of logical implication, verifying that the axiom does not violate any mathematical properties.

Another important feature in formally specifying algorithms and data types is the notion of a co-limit (Smi90). A co-limit is a category theory operation used to combine

algebraic specifications on a shared part (e.g., an algorithm theory and a problem theory) (Smi90). This shared union produces a specialized theory presentation containing features from both *parent* theories. This notion is a powerful concept for composing systems formally.

With the advances achieved in specifying algorithms algebraically, several other researchers are exploring the notion of formally specifying architectures and domains. Gerken hypothesizes that algebraic specifications can formalize the fundamental characteristics of software architectures (Ger93). He is currently developing a method to describe architectures using formally specified notions of components and connectors. Other researchers are working on formalizing domain information. Many believe domains are essentially a collection of "types" (Sri91) that can be formally used, modified, and combined to represent a domain (Bre91). The ability to compose systems and the potential to specify more complex domains make algebraic specifications a powerful notation for defining a software system.

2.3.2 Algebraic Specification Languages. Algebraic specification languages support the development of data type, algorithm, and architecture theories using a three-tuple notation consisting of a set of sort symbols, S, a set of operations, OP, and a set of equations, E (EMCO92):

SPEC = (S, OP, E)

The following subsections examine two algebraic languages in detail, OBJ and LARCH, providing a general understanding of each language's syntax and semantics.

2.3.2.1 OBJ. Similar to the class hierarchy notion in object-oriented languages, the theory behind OBJ is to support the notion of algebraically specified hierarchical models (FGJM85). OBJ is a functional language based on order-sorted equational logic and parameterized programming with an emphasis on the execution of algebraic specifications (Win91). Listed below are the features in the language:

1. *Order-Sorted Algebras:* In OBJ, the notion of order-sorted algebras supports the hierarchical nature of data types. Subtypes (or subsorts) can be defined in a modular fashion. For example, the ordering relationship between natural numbers (i.e., Nat) and non-zero natural numbers (i.e., NzNat) is depicted by NzNat < Nat. This ordering relationship states that NzNat is a subset of Nat (Win91).

2. *Equational Logic:* The operations defined in OBJ are constrained using equational logic. OBJ equations consist of two terms separated by an equivalence connective (i.e., ==). This notation is useful for term rewriting to reduce an equation into its canonical form. Rewriting equations requires finding a unifier that binds a variable to its actual value. This technique closely resembles Lisp's "by value" method (GT79). When an equation can no longer be rewritten, then the expression is said to be reduced or in normal form (FGJM85).

3. *Parameterization:* To enhance the modularity and extensibility of theories, OBJ uses parameterization to generalize specifications and make them more reusable. For example, a list object that is algebraically specified is not tied to a particular type of element, but can be parameterized based on the user's design choice. Thus, parameterization promotes class-based solutions.

```
obj FIBO is
    protecting INT .
    op fibo_ : Int → Int .
    var x: Int .
    eq fibo x = if x < 2
        then x
        else (fibo (x - 1)) + (fibo (x - 2)) fi .
endo
```

Figure 2.5 OBJ Specification of the Fibonacci Computation

4. *Inheritance:* The main object inheritance mechanism used in OBJ is centered around three modes: protected, extended, and using. The protected mode is the most restrictive of the three allowing OBJ modules to use other modules in a protected fashion. That is, changes to the inherited module's operations are not allowed (Win91). On the other end of the spectrum is the using mode which allows OBJ modules to "use" other modules permitting unrestricted changes to the inherited operations. This mode is the loosest of the three and does not guarantee correctness-preserving behavior (Win91). In the middle of this spectrum is the extended mode. The extended mode preserves the operational behavior of the inherited module but allows developers to extend these inherited operations.

OBJ has evolved into a high-level language used to prototype algebraic specifications. Simulating a specification's behavior gives a developer insight into the correctness and coverage of his specification. An OBJ specification defining the computation of the n-th Fibonacci number is shown in Figure 2.5 (Win91).

2.3.2.2 LARCH. LARCH is based on two fundamental principles: clarity of the specification and concrete interfaces between specified programs (GH93). These prin-

```
Table: trait
includes Integer
introduces
  new: → Tab
  add: Tab, Ind, Val → Tab
  __ ∈ __: Ind, Tab → Bool
  lookup:  Tab, Ind → Val
  isEmpty: Tab → Bool
  size: Tab → Int
  0, 1: → Int
  __ + __: Int, Int → Int
asserts ∀ i, i1: Ind, v: Val, t: Tab
  ¬ (i ∈ new);
  i ∈ add(t, i1, v) == i = i1 ∨ i ∈ t;
  lookup(add(t, i, v), i1) ==
    if i = i1 then v else lookup(t, i1);
  size(new) == 0;
  size(add(t, i, v)) ==
    if i ∈ t then size(t) else size(t) + 1;
  isEmpty(t) == size(t) = 0
```

Figure 2.6 Table.lsl

ciples are achieved using two separate representations: a shared language and an interface language.

The LARCH shared language (LSL) is a high-level, generic language that formally describes the fundamental objects, relationships, and properties in a problem domain. The emphasis of the shared language is to describe the observable behavior and the basic constructs of an object. Since LSL traits specify domain-specific concerns, these traits are a principal reusable component (GH93). Figure 2.6 depicts an LSL specification describing the class of tables (GH93).

While the LSL specification captures domain-specific issues, the interface language concentrates on a module's interface and communication to other modules at the implementation level. Based on a particular programming language such as C, Modula, or Ada, the interface language handles implementation dependencies such as parameter pass-

ing and exception handling. This division of labor between the LARCH shared language and the interface language provides a smooth transition from a formally specified problem description into a formal implementation.

2.4 Formal Object Transformation Processes

Both object-oriented methods and formal methods have been praised as successful techniques for specifying software systems (RBP+91) (Sri91). Consequently, research has begun to capitalize upon the successes of both techniques by exploring approaches to integrate object-oriented methods into a formal framework. These efforts are described in the subsequent sections.

2.4.1 A Transformation of OMT Models into a Z Framework. Hartrum and Bailor have developed a formal object transformation process using *Z* as the target model (HB94). They have mapped the OMT object, dynamic, and functional models into a *Z* framework of static and dynamic schemas creating a set-theoretic representation for each object-oriented construct (HB94). The following enumerates Hartrum and Bailor's transformation process:

1. *Object Transformation Process:* A *Z* static schema is developed for each object in the object model. The static schema's signature defines the object attributes over some set-theoretic type (HB94). In the schema's predicate, constraints are written to describe an object's invariant conditions on the object's attributes and between the object's attributes (HB94).

2. *Dynamic Transformation Process:* Z static schemas are also developed to specify the dynamic model with each schema representing an instance of an object's overall state space. Each schema's predicate defines invariants that represents a particular state's variables. Presently, a formal transformation does not exist for translating the entries in a state transition table into a Z representation.

3. Functional Transformation Process: Z dynamic schemas are developed to capture the computational aspect of every functional model. In the dynamic schema's signature, either an object's Δ schema or Ξ schema is included. The Δ schema is included if an object's attributes will be modified during the computation of an object's output values from input values. Otherwise, the Ξ schema is used. The predicate of the dynamic schema describes the actual computation for the object, and it also specifies the invariants that hold during the computation.

Based on the examples provided in the OMT section, Figure 2.7 shows a Z static schema for the Person object class, and Figure 2.8 (HB94) depicts the static schemas for the Person dynamic models. The corresponding state transition table was illustrated in Table 2.1, Subsection 2.2.2. Figure 2.9 (HB94) shows a dynamic schema for calculating a person's age. Additional information on object-orientation and Z can be found in (Wab94).

2.4.2 An Object-Oriented Design Transformation into an Algebraic Framework. While Hartrum and Bailor have created a formal object framework using Z, Breu (Bre91) formalizes object-oriented design constructs using algebraic methods. In Breu's research, the data type is the basic entity during the design phase, and it becomes the cornerstone for her formal transformation process. Breu takes the framework of classes and methods in

Person

$lastname : \text{seq}\, CHAR$
$firstname : \text{seq}\, CHAR$
$middle\_initial : CHAR$
$ssan : \text{seq}\, CHAR$
$sex : SexType$
$age : \mathcal{N}_1$
$birthdate : DateType$

$\#ssan = 9$

Figure 2.7 Static Schema for Person's Object Model.

Child

$p : Person$

$p.age < 18$

Adult

$p : Person$

$p.age \geq 18$
$p.age < 65$

Senior

$p : Person$

$p.age \geq 65$

Figure 2.8 Static Schema for Person's Dynamic Model.

$$
\begin{array}{|l}
\hline
\textit{DetermineAge} \\
\Delta Person \\
\Xi Calendar \\
new\_age! : \mathcal{N}_1 \\
TodaysDate? : DateType \\
Birthdate? : DateType \\
\hline
new\_age! = TodaysDate? - Birthdate? \\
\hline
\end{array}
$$

Figure 2.9 Dynamic Schema for Calculating a Person's Age.

an object-oriented design and maps the framework into an algebraic specification scheme using the notion of inheritance, subtyping, and clientship.

1. *Class Inheritance:* Breu relates the notion of class inheritance in object-oriented design with the notion of vertical composition in algebraic specifications (Bre91). Vertical composition is a step-wise transformation process where the more abstract theory is at the top of the vertical structure. As more details are included in a theory, the abstract theory is refined into a specialized theory. For example, a sort theory describes a generalized notion for arranging elements in a sequence. A more specialized sort theory based on exchanging elements can be specified using a refinement process to produce a shell sort theory (Sri91). The shell sort theory becomes the child in a parent-child theory association. Theories constructed using vertical composition are modular and compact, making them flexible and manageable when specifying large software systems.

2. *Subtyping:* Breu maps type inheritance, i.e., subtyping, in object-oriented designs into an order-sorted algebraic structure (Bre91). Order-sorted algebras are the basis

```
class Set is
        methods
            insert (x: Int) is deferred,
            is_empty return Bool is deferred,
            is_elem (x: Int) return Bool is deferred
            create empty is deferred
end class
```

Figure 2.10 Object-Oriented Class Specification of Set

of a subset relationship between elements, in this case data types. Data refinement is the goal for subtyping, and order-sorted algebras meet this goal.

3. *Clientship:* The object-oriented method stresses the importance of determining the data dependencies among objects, i.e., clientship. Breu relates the persistency characteristic of clientship to the algebraic notion of horizontal composition (Bre91). Horizontal composition refers to the communication between two objects at the same level in a hierarchical model. Since many objects are not mutually exclusive of one another, modeling the correct communication between objects is important. In Breu's algebraic framework, the "uses" clause establishes the horizontal communication mechanism between algebraic specifications.

Inheritance, subtyping, and clientship form the foundation of Breu's transformation process. Breu takes the object-oriented class framework of methods and performs an algebraic mapping into a formal representation. Figure 2.10 shows an object-oriented specification of the class Set. In Breu's algebraic transformation approach, the class methods are mapped directly into algebraic operations with specified domain and co-domain sorts.

```
class spec Set is
    uses<Int, Bool>
    inherits<Spec[parameter: sort]>
    opns
        empty: → Set,
        insert: (Set, Int) → Set,
        is_empty: (Set) → Bool,
        is_elem: (Set, Int) → Bool,
    axioms ∀  s: Set, x, y: Int
        is_empty(empty) = true,
        is_empty(insert(s, x)) = false,
        is_elem(empty, y) = false,
        x = y ⇒ is_elem(insert(s, x), y) = true,
        ¬ (x = y) ⇒ is_elem(insert(s, x), y) = is_elem(s, y)
end class spec
```

Figure 2.11 Algebraic Specification of Set

Each operation's invariants are mathematically modeled as algebraic axioms. Figure 2.11 shows the corresponding algebraic mapping of the class Set.

2.4.3 An Algebraic Transformation of an Unmanned Subway System. Hartrum and Bailor's OMT formalization using *Z* and Breu's algebraic framework for object-oriented design classes are extensive efforts in formalizing the object-oriented process. Dauchy, Gaudel, and Marre take a more informal object transformation approach when algebraically specifying an unmanned subway system.

Since Dauchy, Gaudel, and Marre are only concerned about describing the reactive behavior of the subway, they use state transition diagrams to model two subway software modules. Each transition leaving a state is prioritized and described by a set of boolean conditions. The prioritized boolean conditions are mapped into a set of algebraic equations that describe the valid transitions for a state. Thus, each state transition is represented as

an algebraic equation, and the outgoing state transitions for each state are grouped into a set of algebraic axioms.

Although Dauchy, Gaudel, and Marre's approach demonstrates that state transformation information can be formalized, they fail to describe how to integrate the algebraic axioms into an algebraic specification framework. Consequently, Dauchy, Gaudel, and Marre's approach does not provide a comprehensive framework necessary for an adequate object transformation process.

2.5 Summary

This literature review established a detailed knowledge base to support the development of an algebraic-based execution framework. First, to develop an algebraic transformation process, the algebraic target language should support object-oriented constructs such as inheritance, aggregation, and behavioral constraints. Second, to formally specify OMT analysis models, a systematic approach needs to be developed. This approach must preserve the object-oriented properties of each OMT model and the relationship between the models. Finally, the feasibility of creating an execution framework for algebraic specifications is supported by the advances achieved in using and composing algebraic specifications.

III. Evolutionary Approach to the Formal Object Transformation Process

3.1 Introduction

Chapter I introduced the notion of a formal object transformation process to complement the current KBSE domain application composition system. To make this idea a reality requires establishing a development approach that embodies the major tasks shown in Figure 1.1:

- Creating an algebraic framework for object-oriented models
- Developing an algebraic language compiler
- Producing a unification model for formal languages
- Establishing an execution framework

Because each task produces an incremental product that matures during development, the evolutionary life-cycle was selected as the development strategy. An evolutionary approach uses an iterative process for developing an early operational capability of a system (USA93). Using this iterative process makes the products flexible and easily extensible for future requirements. While each iteration fulfills a minimum set of requirements, together they form the ultimate capability for a system (USA93).

From the four tasks, four phases were defined for the formal object transformation process:

1. An Algebraic Transformation Process
2. An Algebraic Language Parser

3. A Unified Domain Model

4. An Execution Framework

This chapter describes the requirements and design for producing an algebraic framework, the product of the algebraic transformation process. Additionally, this chapter introduces the formal execution stage represented by the last three phases.

3.2 Selecting the Algebraic Framework Language

Selecting a particular algebraic language provided a design foundation and target framework for the algebraic transformation process. Since OMT models are the source for the transformation process, the target algebraic framework must represent the object-oriented characteristics of inheritance, aggregation, state behavior, and the separation of policy and implementation issues. In addition, demonstrating a specification's behavior is an important factor; thus, the ability to execute an algebraic language is the final criterion.

Because of the extensive descriptions provided in Chapter II for OBJ and LARCH, these two languages were selected for evaluation. The following compares and contrasts OBJ and LARCH based on the object-oriented criteria defined above.

OBJ and LARCH are similar with respect to modeling inheritance and aggregation and in providing a well-suited notation for execution. OBJ has three modes of inheritance, protected, extended, or used, that can be selected by the user based on his requirements. LARCH incorporates inheritance using the *includes* and *renames* facilities. For aggregation, LARCH uses the *includes* clause while OBJ uses the *sorts* clause to reference other traits and sorts, respectively. Because OBJ and LARCH are mathematical languages based on

provable constructs, their grammars are precise and well-structured. This precision makes it easy to turn the languages into executable code (NS91). Thus, both languages are well-suited for compilation and execution.

The major difference between the two languages is the syntactical structure of the specification. OBJ is a single-structure specification language that focuses on describing and executing non-deterministic behavior. LARCH'S two-tiered approach makes it a more comprehensive specification language because the approach separates the issues of policy from implementation. Therefore, the two-tiered method allows specifiers to transition from an algebraically-based specification into an algebraically-based implementation. This transition mechanism simplifies the specification of reactive systems by capturing abstract behavior in the shared language while specifying the explicit pre- and postconditions in the interface language.

Another advantage of LARCH is the availability of automated support tools. The creators of LARCH developed a suite of tools to check LSL specifications and to reason about LSL specifications. The wider support environment and the two-tiered approach were the reasons for selecting LARCH as the target language.

3.3 The Algebraic Transformation Phase

Correlating the OMT models described in Chapter II to the target framework, the LARCH shared language, required three, separate algebraic transformation processes: object, dynamic, and functional transformations. The design for each transformation process rested on one principle: preserving the object-oriented constructs found in each model.

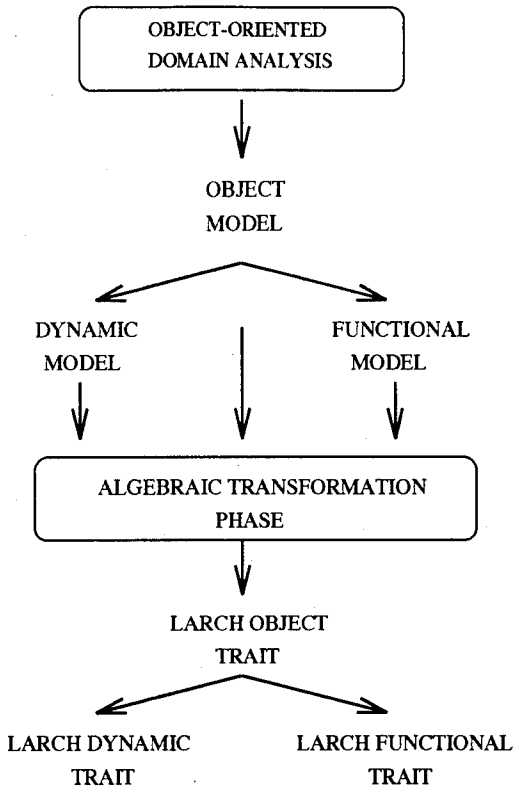


Figure 3.1 Algebraic Transformation Phase

Figure 3.1 illustrates the three algebraic transformation processes, and the following subsections outline the steps required for each transformation.

3.3.1 Object Model Transformation. An OMT object model captures the important objects, object attributes, and relationships between objects in a software system (RBP+91). Its goal is system comprehension, and any algebraic representation of the object model must preserve this goal. Therefore, the object-oriented characteristics of object attributes, attribute invariants, inheritance, and aggregation form the framework for the LARCH object theory.

```
Person: trait
includes Integer
introduces
  [__, __, __]: N, SN, A → P
  __.name: P → N
  __.ssan: P → SN
  __.age:  P → A
  set_name: P, N → P
  set_ssan: P, SN → P
  set_age:  P, A → P
  size: SN → Int
asserts ∀ n, n1: N, sn, sn1: SN, a, a1: A, p: P
  size(p.ssan) == 9;
  ([n, sn, a]).name == n;
  ([n, sn, a]).ssan == sn;
  ([n, sn, a]).age == a;
  set_name([n, sn, a], n1) == [n1, sn, a];
  set_ssan([n, sn, a], sn1) == [n, sn1, a];
  set_age([n, sn, a], a1) == [n, sn, a1]
```

Figure 3.2 Attributes Transformed into Algebraic Operators

Transforming the object classes into a LARCH object theory framework requires the following steps:

1. Create an LSL trait for each object class.
2. Transform each object's attribute in one of two ways:
   - Create an LSL operator for each object attribute. For each operator, define the appropriate signature specifying the attribute's domain and range. Figure 3.2 depicts this alternative using the Person class.
   - Create an LSL sort representing the trait and use the tuple notation. Each tuple field represents an attribute of the object. This alternative is illustrated in Figure 3.3.

```
Person: trait
includes Integer
  P tuple of name: N, ssan: SN, age: A
introduces
  size: SN → Int
asserts ∀ p: P, n: N, sn: SN, a: A
  size(p.ssan) == 9;
  p.name == n;
  p.ssan == sn;
  p.age == a;
```

Figure 3.3 Attributes Transformed into an Algebraic Tuple Notation

3. Because derived attributes are computed from static attributes, LARCH's operator notation best captures derived attributes. Therefore, for every derived attribute, create an LSL operator with the appropriate signature.

4. Use axiomatic equations to define constraints on attributes and between an object's attributes. Also, use algebraic equations to specify an operator's behavioral constraints.

5. Analyze the inheritance and aggregation properties of the object model. For each object that inherits from its parent, include the associated parent trait in the LSL *includes* clause. To specialize a parent trait's parameter values, rename the parameters using LARCH's *rename* notation. For each object that uses another object, provide the used object trait in the LSL *includes* clause.

6. Reserve multiple inheritance for future efforts. Although, multiple inheritance is an important feature in object-oriented models, it increases the complexity of algebraic specifications. Therefore, the initial design of the algebraic object theory includes only single inheritance.

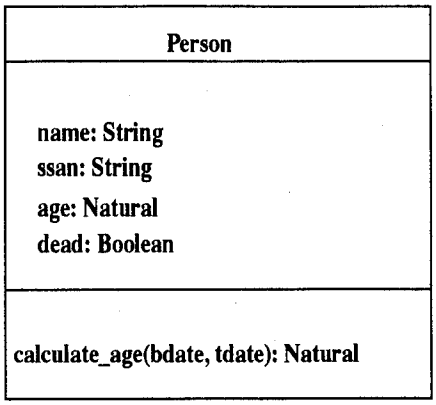


Figure 3.4 Person Object Model

```
Person: trait
includes Integer
  P tuple of name: N, ssan: SN, age: A
introduces
  size: SN → Int
asserts ∀ p: P, n: N, sn: SN, a: A
  size(p.ssan) == 9;
  p.name == n;
  p.ssan == sn;
  p.age == a;
```

Figure 3.5 Person Object Trait

To illustrate the object transformation process, Figure 3.4 shows an object model for a Person class, and Figure 3.5 represents its formal framework.

3.3.2 Dynamic Model Transformation. The most challenging of the three transformation processes is the dynamic model transformation. The OMT dynamic model describes the reactive behavior of an object using states and events. Therefore, the dynamic transformation process must preserve two important characteristics: an object's state space and an object's state transition information (i.e., events, current state, and next state).

To use the LSL traits as a framework for the OMT dynamic model required evaluating two different approaches: Dauchy, Gaudel, and Marre's approach and the Mealy Machine approach. The criteria for evaluation was to select an alternative that mapped best to the OMT dynamic model and to the LSL trait characteristics.

Dauchy, Gaudel, and Marre demonstrate one alternative for formally modeling state transitions in their subway system example. They use boolean expressions to model each state transition, prioritizing each state's boolean expressions to form a set of axioms. Even though their approach transforms the state transitions into a set of algebraic axioms, they fail to integrate these axioms into a coherent algebraic theory.

The Mealy Machine model provides a second alternative. It associates each output of a state with a state transition. In relation to the OMT dynamic model, this corresponds to associating each output of a state with an activity. That is, the input to an activity would be a state of an object and the activity's output determines the next state of an object. Formally modeling the states and activities is possible within the LARCH framework. Therefore, the Mealy model fits in well with the OMT models and the LARCH algebraic structure.

The following steps describe the dynamic transformation process using the Mealy Machine approach:

- LARCH *Framework for States:*

    1. Transform each state in a dynamic model into a separate Larch trait that *includes* the object class trait.

2. For each state trait, define a valid-state boolean operator. The operator's signature defines the object as the domain and a boolean as the range. The operator determines whether or not an object is within a particular state.

3. Analyze each valid-state operator and define appropriate behavioral constraints in the LSL asserts clause. The constraints represent the valid attribute ranges for a state and are in the form of axiomatic equations.

- LARCH *Framework for Events:*

1. For each event in the dynamic model, define an event trait. In each event trait, use boolean operators to specify the event.

2. Analyze each event operator and define appropriate constraints in the LSL asserts clause.

3. Reserve the activities associated with events for the functional transformation process.

- LARCH *Framework for State Transition Tables:* An object's state transition table identifies an object's event sequencing by documenting the object's current state, external event, and next state relationship. Specifying these two subspaces (i.e., the current and next states) in an algebraic equation violates the consistency property. For example, the equation:

$$child(P) \wedge birthday(age = 17) => adult(P)$$

specifies when a Person object "P" transitions from a child state into an adult state. This equation is inconsistent because the implication states that a Person can be in

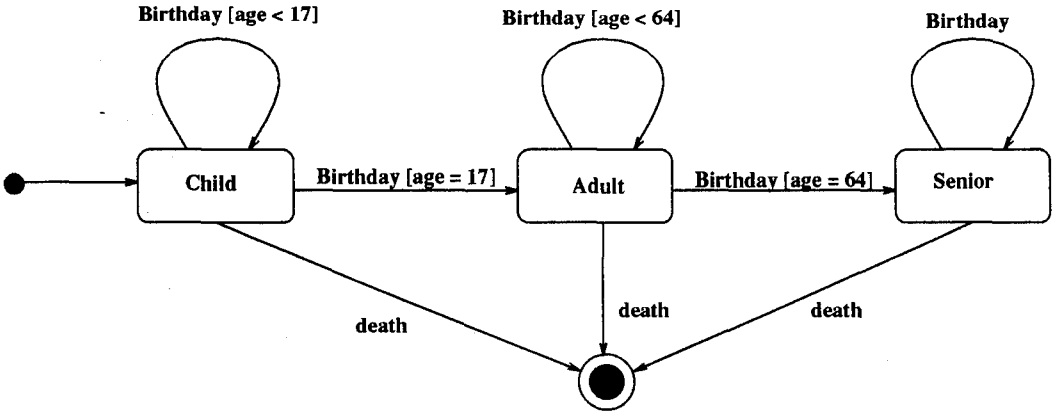


Figure 3.6 Person's Age Dynamic Model

two different states at one time (i.e., in the child and adult state simultaneously). This inconsistent equation cannot be incorporated into an LSL specification. Thus, the most effective way to specify an object's event sequencing is to retain Rumbaugh's state transition table notation for each object class. To complete the dynamic transformation process, create a State Transition Table (STT) for each dynamic model and include it with the dynamic theories.

The following figures demonstrate a complete dynamic transformation process for the Person class. Figure 3.6 shows a dynamic model of a Person's age states, Figure 3.7 shows the corresponding LARCH state traits, Figure 3.8 specifies the LARCH event trait, and Table 3.1 defines the state transition table.

3.3.3 Functional Model Transformation. An OMT functional model describes an object's data transformations, i.e., a collection of data and operations on the data, which is equivalent to an abstract data type. Thus, encapsulating a functional model in a LARCH

```
ChildState: trait
includes Person
introduces
  child: P → Bool
asserts  ∀ p: P
  child(p) == p.age < 17;

AdultState: trait
includes Person
introduces
  adult: P → Bool
asserts  ∀ p: P
  adult(p) == p.age < 64 & p.age ≥ 17;

SeniorState: trait
includes Person
introduces
  senior: P → Bool
asserts  ∀ p: P
  senior(p) == p.age ≥ 64;
```

Figure 3.7 Person State Traits

```
Birthday: trait
includes Person, Calendar
introduces
  birthday: P, C → Bool
asserts  ∀ p: P, c: C
  birthday(p, c) == p.bdate = c.tdate;
```

Figure 3.8 Person Event Trait

Table 3.1 Person State Transition Table.

Current State	Event	Parameters	Guard	Next State	Action
Child	Birthday		age < 17	Child	
Child	Birthday		age = 17	Adult	
Child	death			Terminated	
Adult	Birthday		age < 64	Adult	
Adult	Birthday		age = 64	Senior	
Adult	death			Terminated	
Senior	Birthday			Senior	
Senior	death			Terminated	

specification is straightforward. Defined below are the three steps for the transformation process:

1. Map each data transformation into a separate Larch trait. Include the object trait.
2. Define an LSL operator that models the transform process. Model the inputs and output of the transform function in the operator's signature.
3. Model the behavioral constraints associated with the transform as LSL algebraic equations.

To complete the Person class example, Figure 3.9 shows a functional model for calculating a person's age, and the corresponding LARCH functional theory is depicted in Figure 3.10.

3.3.4 Validation. To effectively demonstrate the algebraic transformation process, two validation domains were selected: the counter and fuel tank domains. The counter domain represents a simple non-reactive system used to test the initial design concepts of the object and functional transformations. The fuel tank domain is a small-scale reac-

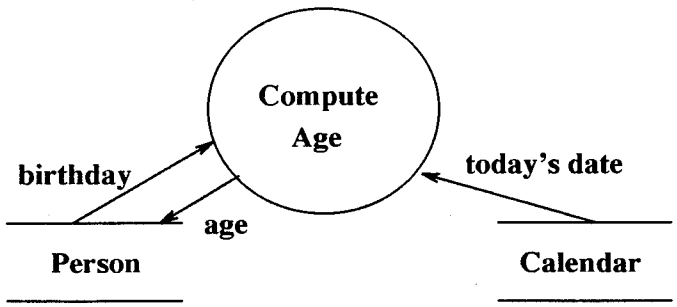


Figure 3.9 Functional Model for Computing a Person's Age

```
CalculateAge: trait
includes Person, Calendar
introduces
  NewAge: C, P → Int
asserts  ∀ p: P, c: C
  NewAge(c, p) == [p.name, p.ssan, c.tdate - p.bdate, p.bdate]
```

Figure 3.10 Functional Trait for Computing a Person's Age

tive system that provides a more rigorous exercise of the entire algebraic transformation process. This domain contains a complete set of object, dynamic, and functional models developed and refined by Hartrum and Bailor (HB94). The validation criteria levied on both domains centered around two principles:

1. Coverage - The portion of OMT Analysis Models successfully formalized.
2. Consistency - An object's behavioral constraints do not contain contradictions (i.e., true = false).

The following subsections describe each domain's adherence to these principles.

3.3.4.1 Counter Domain. The counter domain contains a counter object class and three data transformations: increment, add, and subtract, depicted in Figure 3.11. Thus, each of these constructs must be formally modeled and validated. The following

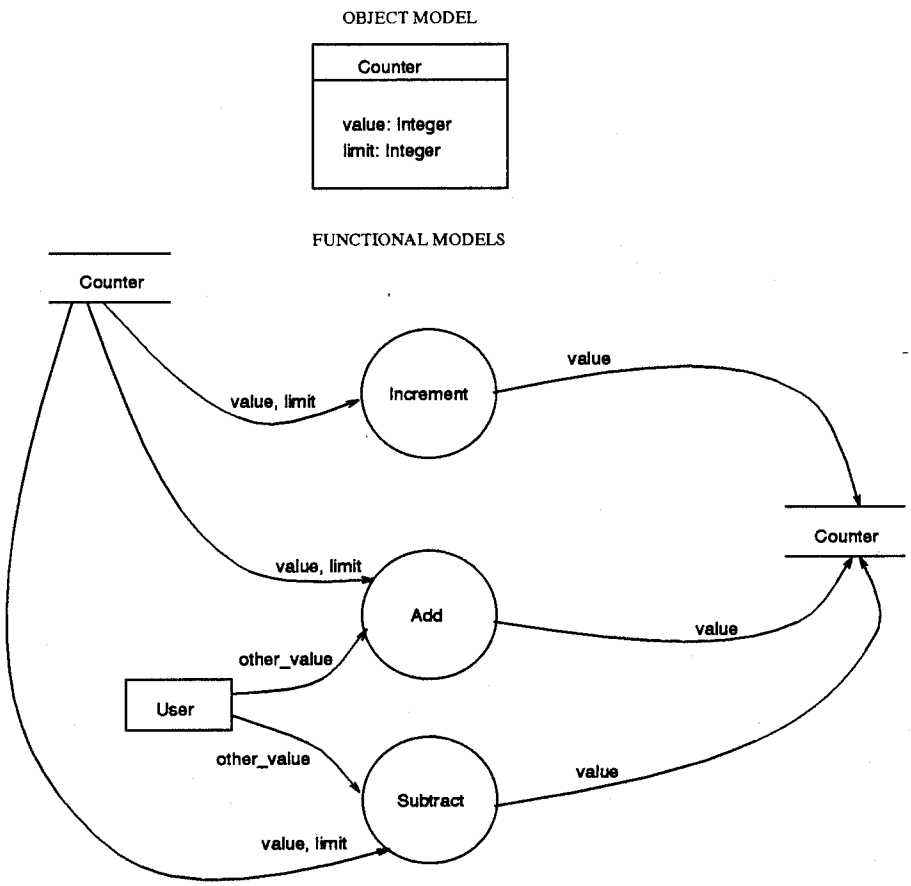


Figure 3.11 Counter Domain's OMT Models

describes the validation process:

1. *Formal Framework Development:* A formal framework was established for the counter domain consisting of four traits: an object trait and three functional traits. The object trait represents the counter object class capturing the object attributes in a tuple notation and specifying the invariants between the attributes using axiomatic equations. Each functional trait models a transform function as an abstract operator and uses axiomatic equations to represent the behavioral constraints. Therefore, the entire domain was formally modeled, conforming to the coverage criteria. This formal framework is found in Appendix A.
2. *Domain Theory Evaluation:* The traits were evaluated against the original domain model to ascertain its consistency with the model's object structure, operations, and invariants. This evaluation demonstrated that the domain theory precisely identified the important properties and behavioral constraints in the counter domain.

3.3.4.2 Fuel Tank Domain. The fuel tank domain consists of an object class with several attributes and a set of four states and events. Because there are numerous functional transformations in the domain, three were selected as a subset for validation purposes: Determine Interval, Calculate Filled Level, and Predict Tank Full Time. Appendix B contains the OMT models and state transition table for the Fuel Tank Domain. The following enumerates the validation results:

1. *Formal Framework Development:* Analogous to the counter domain's validation process, a formal framework was established for the fuel tank object class and transform

functions using the same tuple notations and operator constructs. Therefore, the object and functional transformation processes conformed to the coverage criteria. Coverage for the dynamic model transformation was defined as formally specifying the states and events with their invariants, and the dynamic transformation process fulfilled this objective. The formal specifications for the Fuel Tank Domain can be found in Appendix C. In addition, Appendix D contains a REFINE object-based description for the state transition tables. This specification parses the Fuel Tank's state transition table creating objects for the execution framework.

2. *Domain Theory Evaluation:* The LARCH framework provided a precise description of the fuel tank domain by capturing the domain's essential properties and behavior. Each property could be checked systematically because of the specification's precise notation. Thus, this transformation validated the consistency criteria.

3.3.5 Summary. This section described the algebraic transformation process and presented the validation criteria. Validating the transformation process proved extremely beneficial for testing the coverage and consistency of the transformation process. Satisfying these two criteria demonstrated that the OMT models could be formally transformed into an algebraic framework without loss of information (i.e., the structural and behavioral properties were preserved). The operational products from this phase consist of a set of LARCH traits representing a domain theory. The next phase uses these domain theories as input for generating an initial execution framework for formal specifications.

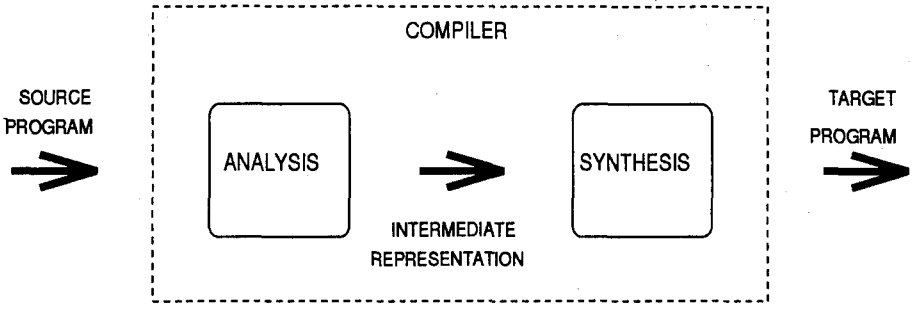


Figure 3.12 Analysis-Synthesis Model

3.4 The Formal Execution Phase

The formal execution phase is responsible for translating the LARCH specifications into executable programs. Figure 3.12 depicts two distinct stages in creating a compiler: analysis and synthesis of the source language (ASU88). For the formal object transformation process, the compilation process is extensive; therefore, three chapters have been devoted for describing the details of each compilation evolution. Chapter IV describes the design and development of a LARCH parser. Chapter V presents a unified approach for developing LARCH and *Z* language parsers. Finally, Chapter VI contains a design for an execution framework.

3.5 Summary

This chapter established a basis for using an evolutionary development approach, evaluated two algebraic language representations, and enumerated the steps for an algebraic transformation process between OMT models and LARCH traits. An evolutionary approach was selected because of the iterative nature of developing a formal transformation process. The first iteration required selecting an algebraic target language to support the design of an algebraic transformation process. LARCH was selected because of its two-tiered

specification approach and available support environment. The algebraic transformation process specified a set of steps required to transform the OMT models into a LARCH framework of traits. Based on preserving the object-oriented characteristics in the OMT models, each step ascertained the resulting domain theory's consistency with the domain model. Consistency and coverage were the two validation criteria used to validate the transformation process. The next phase uses the validated domain theories as input for producing a robust LARCH parser, the first step towards developing a formal execution framework.

IV. Design and Implementation of a LARCH *Parser*

4.1 Introduction

After establishing LARCH as our algebraic framework, the next step focused on developing a robust LARCH parser. Language parsing, depicted in Figure 4.1, combines the lexical and syntactical analysis processes together to generate a syntax tree. First, the lexical analysis phase scans a source program and groups the characters into a set of meaningful tokens. Then, the syntactical analysis process combines the tokens to create a program's syntactic hierarchy. A well-designed parser simplifies the semantic analysis tasks leading to a more robust intermediate representation for code generation (ASU88).

This chapter presents two iterations in the design and implementation of a LARCH parser. It describes the parsing environment, outlines the requirements for the parser, and discusses the design decisions and trade-offs. In addition, the validation results are presented.

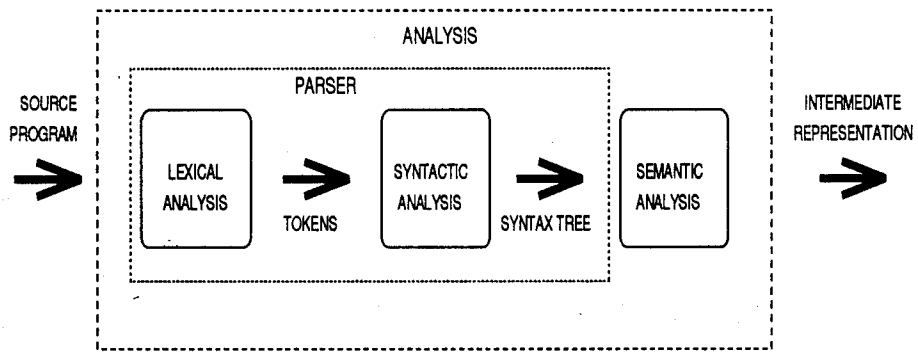


Figure 4.1 Compilation Analysis Phase

4.2 SOFTWARE REFINERY - *The Parsing Environment*

Developing a language parser requires a stable development environment. SOFTWARE REFINERY was selected because it offers a complete programming environment to support not only language parsing but also code generation. The following lists some of the strengths of the SOFTWARE REFINERY environment (Rea):

- It provides capabilities to formalize specifications and to generate executable programs from these specifications.
- It supports a wide range of specification techniques from implicit transformations to explicit programming constructs like *if-then-else.*
- It is an object-based environment that allows specifiers to query and manipulate objects.
- It contains a powerful toolset that supports language parsing.

Because this chapter emphasizes parsing, we concentrate on SOFTWARE REFINERY'S parsing toolset: DIALECT and OBJECT INSPECTOR.

DIALECT allows one to construct a formal language syntax using the notion of object classes and map attributes in an extended Backus Naur Form (BNF) notation. The object classes represent the unique non-terminals on the left-hand side of a language's grammar rules, and the map attributes form the corresponding right-hand side. These object classes and attributes define the domain model for a language in REFINE'S object base. Once a grammar and domain model are constructed, DIALECT can parse source programs, storing them in the object base as abstract syntax trees (ASTs) (And92). These ASTs can be

viewed using OBJECT INSPECTOR, a graphical interface tool, that allows one to discern the underlying structure of a source program.

4.3 LARCH *Parser - Part I*[1]

There was one fundamental goal for developing a LARCH parser - to represent and parse the complete language. Meeting this objective required developing a stable domain model and grammar in REFINE'S object base. The original LARCH grammar, developed at MIT, was the primary reference for identifying the core objects and associations.

4.3.1 Domain Analysis. LARCH'S modular structure makes a top-down decomposition of the grammar rules into object classes and map attributes a suitable approach. The following summarizes the main object classes in LARCH and Figure 4.2 graphically depicts the top-down decomposition.

1. *Trait Name:* Identifies the specification.
2. *Trait Parameters:* Sorts used as parameters that generalize a specification, making them reusable. For example, a list specification can generically designate a sort E to represent any valid element type. Then, this sort can be specialized based on a user's requirements (e.g., renaming E to Integer).
3. *Trait Body:*
   - *Context References:*

[1]The original LARCH domain model and grammar described in the following section are not included in this document. The final versions are presented in Chapter V, the unification phase.

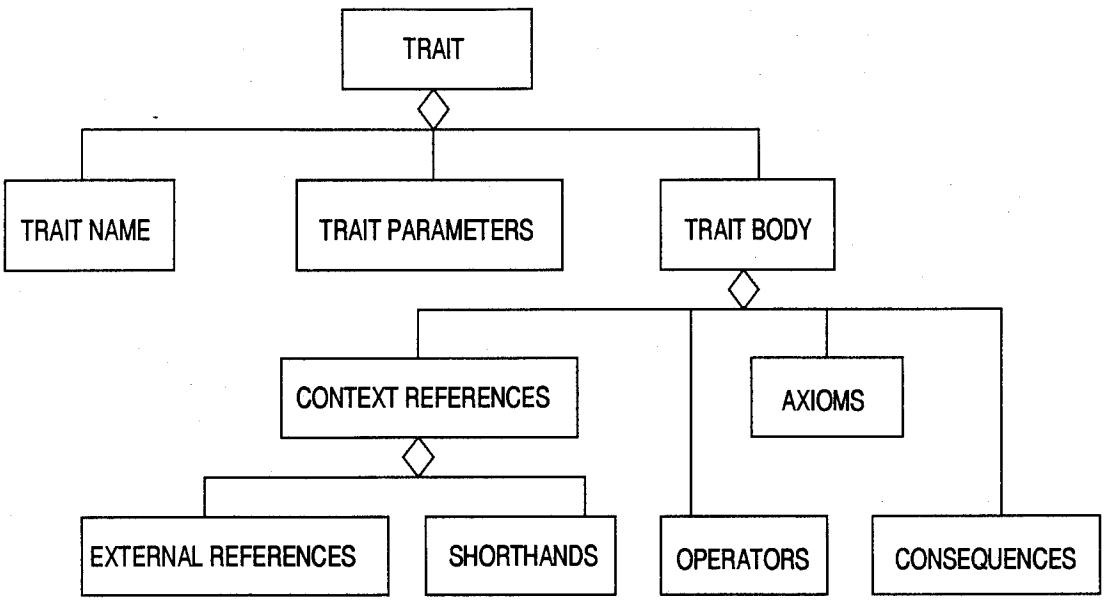


Figure 4.2 Main Object Classes in LARCH

- *External References:* Allows a specification to reference another specification using an *includes* or an *assumes* association.
- *Shorthand Notations:* Compact notations used to shorten and simplify a LARCH specification. They include tuples, enumerations, and unions (GH93).

- *Operators:* Represent total functions that define a specification's abstract operations. Each function contains a signature denoting its domain and range sorts (GH93).
- *Axioms:* Algebraic equations used to define the behavior of operators and relationship between operators.
- *Consequences:* Specify redundant algebraic equations that provide checkable properties (i.e., consistency, completeness, and correctness).

4.3.2 Parser Validation. After defining the entire grammar in the object base, two domains were selected for validation: a sample set of traits developed at MIT[2] and the counter domain. The MIT traits were selected because they have been rigorously examined by the LARCH developers. In addition, these traits covered a myriad of LARCH language features and exercised many of the capabilities needed in the parser. The counter domain was selected because it continued the domain validation process initiated in Chapter III.

Both test domains were evaluated using three criteria: coverage, consistency, and correctness (Wab94). To adhere to these criteria, two test cases were established.

1. *Test Case 1 - Grammar Validation:* This test is responsible for identifying and resolving the ambiguity in the grammar during compilation. The parser reports these conflicts as either shift/reduce or reduce/reduce errors. That is, a shift/reduce conflict occurs when the parser does not know whether to shift a new token onto the current sentence, or reduce the tokens into a sentence. On the other hand, a reduce/reduce error happens when the parser has two reduction choices.

   In our case, ambiguity occurred while compiling the theorem proving equations because the equation's production rules matched the rules for the axiomatic equations. An analysis of the debug file failed to identify a solution to these errors. Therefore, to produce a rapid prototype of the parser, the theorem proving clauses were omitted since they only added redundancy checking. Removing these clauses resulted in less than 100% coverage of the language.

[2]The Massachusetts Institute of Technology (MIT) has a public domain LARCH site that contains a compendium of LARCH information, including a collection of LARCH traits. This site can be accessed through world-wide web, http://larch-www.lcs.mit.edu:8001/larch/

2. *Test Case 2 - Parser Evaluation:* Parser evaluation consisted of analyzing the abstract syntax trees against the grammar's syntactic rules to demonstrate consistency and correctness. During the first pass, we discovered the ASTs violated the grammar's syntax. This occurred because the parser did not enforce the proper selection when encountering an *or* (rule choice) notation in the production rules. Correcting this error resulted in a parser that was correct and consistent with the original grammar's syntax rules.

4.3.3 Summary. Although the traits parsed successfully, OBJECT INSPECTOR showed that the ASTs contained several recursive layers making the ASTs difficult to understand. At times, OBJECT INSPECTOR could not bring up the entire AST because of the AST's unwieldy structure. Consequently, instead of providing the desired concise description of LARCH'S syntax, the ASTs were too complex.

4.4 LARCH *Parser - Part II*[3]

The prototyped parser defined in Section 4.3 generated ambiguous ASTs and proved to be an inadequate parser because it did not satisfy the coverage criteria. Specifically, LARCH'S grammar rules for defining algebraic equations were too abstract. In addition, the complete LARCH language was not implemented due to the ambiguity of the theorem proving clauses. Overcoming these deficiencies required a second iteration of the parser. This iteration established two goals: to parse the entire LARCH language and to generate

[3]Again, because of the evolutionary development approach, the original model and grammar are not documented. The final versions are presented in the next chapter.

well-defined ASTs. In order to generate precise ASTs, a domain analysis was required to identify the areas of abstraction.

4.4.1 Domain Analysis. A new domain model for the LARCH language was constructed beginning with the object classes defined in Subsection 4.3.1 and shown in Figure 4.2. The original grammar contained several recursive hierarchies because the LARCH developers wanted to provide an abstract description of the language emphasizing brevity rather than executability, but, we required a more concrete syntax to implement the grammar. Therefore, two new sub-hierarchies were defined to make the grammar more explicit: an algebraic equations and a consequences equations hierarchy.

4.4.1.1 An Algebraic Hierarchy. The original grammar represented algebraic equations in an abstract manner. Figure 4.3 illustrates the definition of the addition expression "$arg1 + arg2$". This figure clearly depicts the recursive nature of the grammar and the onerous AST representation.

To reduce the amount of recursion, each abstract expression was decomposed into individual expression types. That is, the domain model represented each algebraic expression (e.g., add, subtract, and multiply) as individual objects (i.e., subclasses) of the algebraic expression class. This concrete definition generated a wider AST structure but reduced the AST's depth and complexity. Figure 4.4 demonstrates a more precise representation of the add expression described above.

4.4.1.2 A Theorem Proving Hierarchy. Recursion in the theorem proving clauses caused similar ambiguity errors in the initial parser. The ambiguity occurred be-

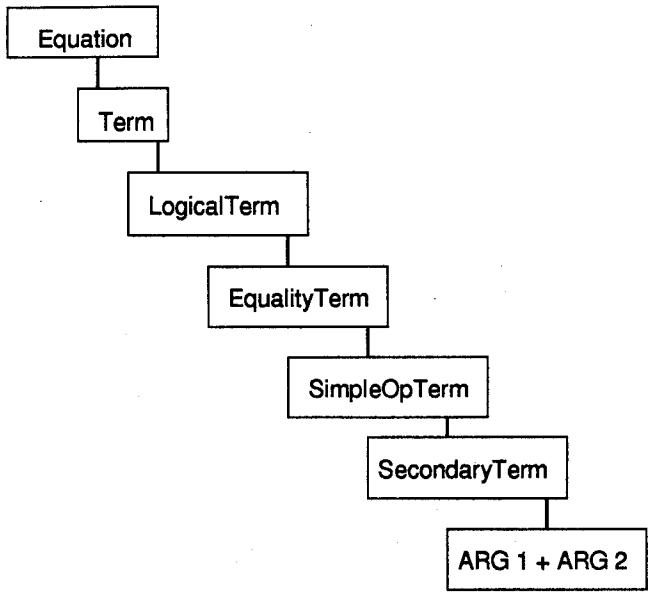


Figure 4.3 Recursive Nature of LARCH

cause the LARCH theorem proving definition contains a choice between two similar rules: one that contains an equations formulation and another that contains a quantified equations formulation. The similarities and recursive nature of these rules confused the parser, making the parser unable to resolve the ambiguity. To solve this problem, two separate object classes were defined to represent each choice. This approach eliminated one level of recursion, clarified the parsing choices, and simplified the corresponding AST structure.

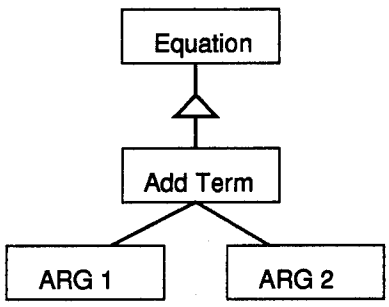


Figure 4.4 Concrete Definition of an Algebraic Expression

4.4.1.3 Domain Model Summary. The completed object-oriented domain model proved to be an ideal map into REFINE's object base. The objects and associations mapped directly into object classes and map attributes forming the class hierarchy and surface syntax for the LARCH parser. After defining the domain model and grammar in REFINE, the parser was ready for evaluation.

4.4.2 Parser Validation. This validation process paralleled the first one using the same test sets, validation criteria, and test cases. The results obtained from the validation process are listed below.

1. *Case 1 - Grammar Validation:* During compilation, DIALECT reported one reduce/reduce error. This error occurred in the LARCH consequences section. Since only one error occurred, analyzing this problem was reserved for the parser evaluation phase to determine the severity of the error.
2. *Case 2 - Parser Evaluation:* This evaluation revealed that the grammar omitted several algebraic expressions; some of them are depicted in Figure 4.5. This occurred because of the explicit nature of defining algebraic expressions. Although the terms were initially omitted, adding these expressions was straightforward because of the well-structured domain model and grammar designed in Section 4.4.1. After adding the undefined expressions, the single reduce/reduce error was evaluated. A visual inspection showed that the ASTs matched the original grammar's syntactic structure; thus, the single reduce/reduce error did not adversely impact the parser.

```
asserts ∀ a: A, l, l1, l2: C
  flatten((atom(a) ⊣ l)) == atom(a) ⊣ flatten(l);
  flatten((list(l1) ⊣ l2)) ==
    flatten(l1) || flatten(l2);
  reverseAll((atom(a) ⊣ l)) ==
    reverseAll(l) ⊢ atom(a);
  reverseAll((list(l1) ⊣ l2)) ==
    reverseAll(l2) ⊢ list(reverseAll(l1));
```

Figure 4.5 Example of Undefined Axiomatic Terms

4.4.3 Summary. The end of the second parser evaluation satisfied all three validation criteria:

1. *Coverage:* Generating a parser for the complete language
2. *Consistency:* Remaining consistent with the grammar's syntax
3. *Correctness:* Producing syntactically correct ASTs

4.5 Summary

The goal to develop a robust LARCH parser was achieved during this evolution. Reaching this goal required redefining several ambiguous language expressions to generate well-structured ASTs. An unambiguous AST simplifies the remaining compilation process by providing a stable framework for performing semantic analysis tasks and code generation. In addition, a robust LARCH parser establishes a strong foundation for the next phase, developing a unifying abstract framework for formal languages.

V. *Design and Implementation of a Unified Domain Model*[1]

5.1 Introduction

In Chapter I, two complementary paths, a set-theoretic, or model-based approach and an algebraic, or theory-based approach, were presented for formalizing object transformations into the REFINE object base. Figure 5.1 depicts this transformation process and also shows a third horizontal path. This path represents another transformation mechanism, one intended to produce a unified model of the designated formal specification languages, *Z* and LARCH.

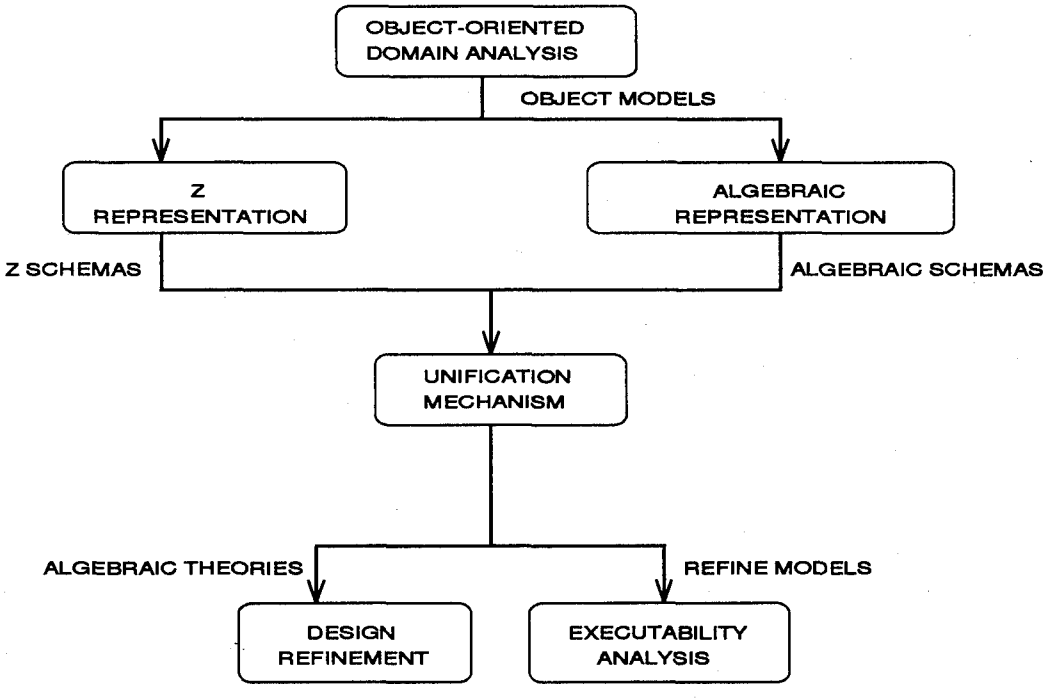


Figure 5.1 Formalized Object Transformations

In order to automate a portion of the first transformation process, both vertical paths contain a step for language-specific compilation. Providing an initial operational capability

[1]This chapter was co-written with Captain Kathleen Wabiszewski. It also appears in (Wab94).

toward this goal, robust language parsers were designed and implemented. These parsers generated similar structural representations (abstract syntax trees) that established a basis for the preliminary analysis and design of a unified target model.

This chapter focuses on the evolution of a Unified Domain Model, whose aim is the creation of a single, cohesive framework for translating different source specifications into a unifying abstract structure. Four iterations define this evolution, and they are detailed in Sections 5.2 through 5.5:

1. Evaluation of Abstract Syntax Trees (AST)
2. Analysis of Design Alternatives
3. Development of a Unified Core Model
4. Development of Language Specific Extensions

5.2 Evaluation of Abstract Syntax Trees

As mentioned in Section 5.1, the parser-generated ASTs provided a common focal point for analyzing the structures of LARCH and *Z* to establish the design of a unified target model. This evaluation process was likened to an electronic balanced mixer circuit. Accepting two input frequencies, a balanced mixer produces four outputs: the two original frequencies, the sum, and the difference. Considering the two ASTs as input, the evaluation produced the originals, common core objects, and language specific objects, as depicted in Figure 5.2.

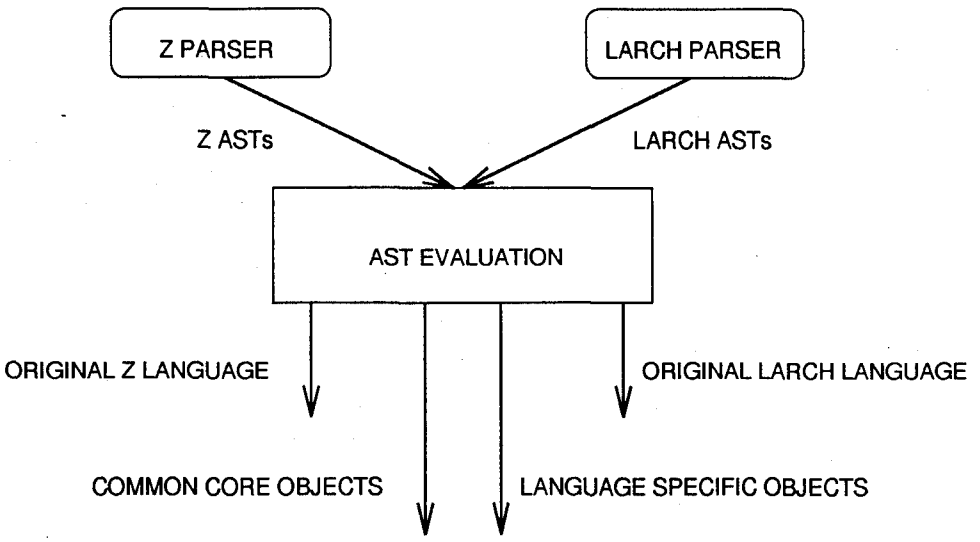


Figure 5.2 AST Evaluation Process

5.2.1 Common Core Objects. The evaluation of the ASTs revealed that both languages contain strong similarities in their conceptual representations of a specification. Both languages require a set of signatures, axioms, and external references to describe a problem domain. LARCH uses operators with signatures, axiomatic equations, and context references in its specification format. *Z* defines a specification using signatures, predicates, and schema referencing. Even though the syntactical domain, or notations, differ, the semantic domain, i.e., the mathematical foundation, remain fundamentally the same. Consequently, Table 5.1 identifies a set of core objects extracted from both languages that form the unified domain model.

5.2.2 Language Specific Objects. The AST analysis also revealed differences in the way each language characterized the common objects. For example, even though the ASTs identify that both languages contain a signature, a *Z* signature can use specific notions of state transitions, both the *before state* (State) and the *after state* (State'), while

Table 5.1 LARCH and *Z* Commonalities

Unifying Object	LARCH	*Z*
Theory Object	Trait	Schema Paragraph
Theory Id	Trait Id	Schema Id
Theory Body	Trait Body	Schema
Theory Signature	Introduces Clause	Declarations
Theory External Reference	Context References	Schema References
Theory Axioms	Asserts Clause	Axiom Part

LARCH does not. Additionally, the *Z* language is capable of explicitly declaring input and output variables, while LARCH cannot. Therefore, these variances required an approach that captured each language's specialization, yet preserved the core characteristics.

5.2.3 A Framework for Language Inheritance. One approach for modeling the common core objects and the specialized objects is through language inheritance. Language inheritance establishes a notion of a common base language that is inherited by various *dialects* (Sys90). These dialects specialize the common language in order to implement language specific constructs. This approach seemed well-suited for modeling the specialized constructs of LARCH and *Z* because each language can be represented as a *dialect* of a common core language, i.e., a mathematically-based language composed of signatures, axioms, and external references. Establishing this framework in REFINE supports the development of a uniform interface for formalized object-oriented models. The REFINE object base is capable of representing inheritance and specialization through object classes and subclasses. Figure 5.3 illustrates an example of language inheritance for LARCH and *Z* using the unifying objects defined in Table 5.1.

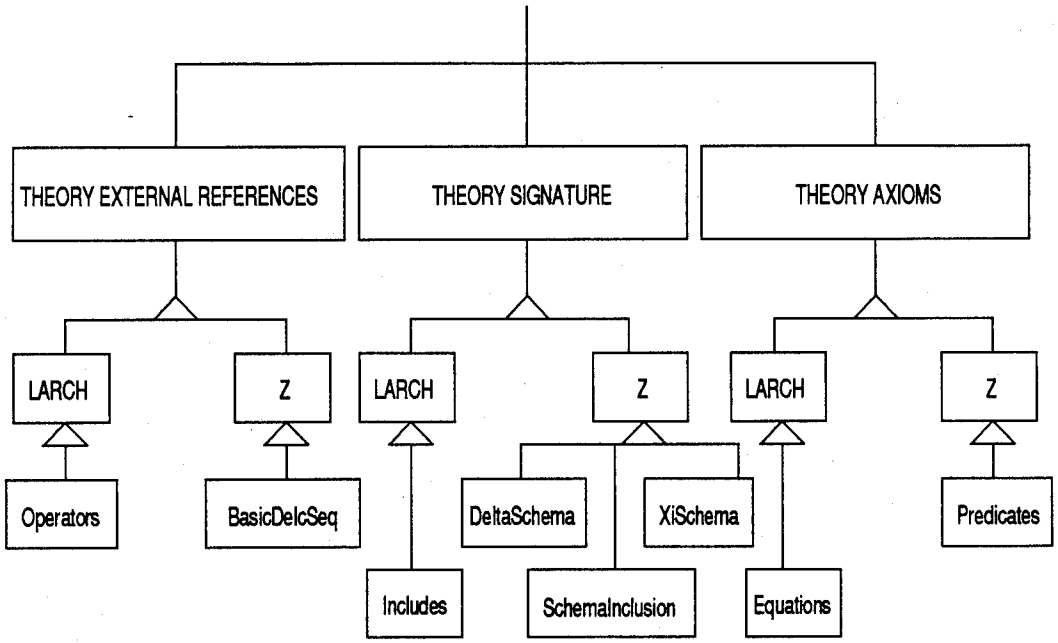


Figure 5.3 Language Inheritance

5.3 Analysis of Design Alternatives

Using language inheritance as a framework to consolidate the similarities and differences between LARCH and *Z* modified the initial transformation process depicted in Figure 5.1. Instead of converging to a unified model after compilation, the new design focused on unifying the two languages during the parsing stage. This new process is founded on the common base language established above and is shown in Figure 5.4.

In order to implement this new design, two possible courses of action were developed:

1. Develop transformation programs for each language to convert the grammars into the unified model.
2. Refine the original language grammars in order to parse into the unified domain model.

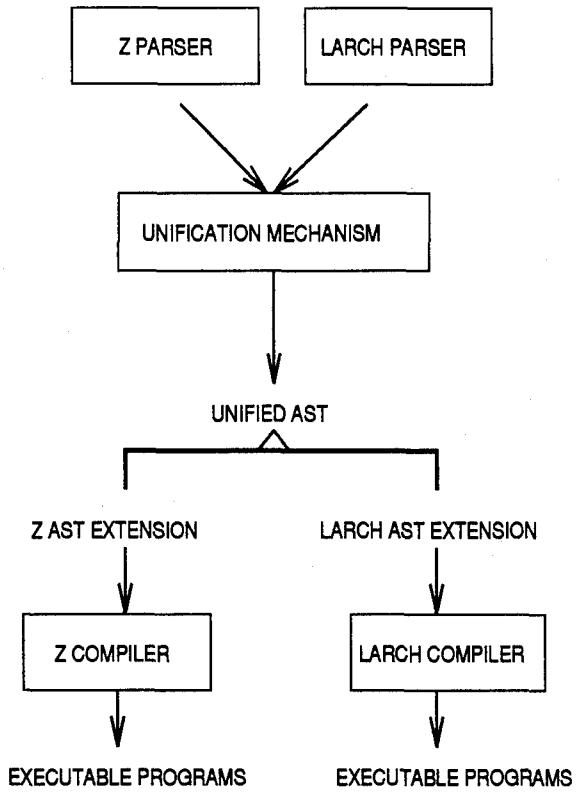


Figure 5.4 Modified Transformation Process

Since these paths were significantly different, an in-depth analysis was conducted to select the best candidate. The strengths and weaknesses of each alternative are summarized in Figure 5.5 and discussed in the two subsequent sections.

5.3.1 Transformation Approach. Even though the transformation approach decreases the parser's complexity, the process introduces a level of informality into the unified implementation. This informality results from relying on hand-coded algorithms to construct the unified model instead of a more established parsing technique that relies on machine-generated algorithms. Also, since reusable code is essentially prohibited, the transformation approach increases the amount of transform specializations needed in the

UNIFICATION APPROACH ALTERNATIVES

	ADVANTAGES	DISADVANTAGES
TRANSFORM APPROACH	LESS COMPLICATED COMPILERS MORE EXECUTION MODEL ANALYSIS	ONE MORE LEVEL OF TRANSLATION INCREASED TRANSFORMATION SPECIALIZATIONS
PARSING APPROACH	ONE LESS LEVEL OF TRANSLATION TRULY COMMON CORE POSSIBLE SET OF WELL-DEFINED REFINE ABSTRACTIONS	MAY COMPLICATE COMPILERS SACRIFICE WORK ON EXECUTION MODEL

Figure 5.5 Unification Design Alternatives

overall formal process. More importantly, despite the advantage of completing more analysis of the execution model, this model does not adhere to the desired correctness constraint.

5.3.2 Parsing Approach. The parsing approach complicates the development of the formal language compilers, thereby placing a time constraint on performing a complete execution model analysis. But, this approach preserves each languages' syntactical structure in the unified implementation. By using DIALECT to accept both the unified domain model and each language's surface syntax, DIALECT generates a parser for Z and a parser for LARCH that map to the same common core model. Because this approach preserves each language's syntax, thereby fulfilling the correctness requirement, it is the strongest candidate for the unification implementation.

5.4 *Development of a Unified Core Model*

After completion of the AST evaluations and approach analyses, the next iteration in the evolution of the unified domain model focused on the common core objects. Based on previous experience with the original parsers and the sensitivity of the DIALECT tool, an incremental strategy was adopted to reduce the overall complexity of this development phase. Using the three OOA models as increments, the following steps were outlined to parse both LARCH and *Z* into the unified core:

1. Design a Unified Core Domain Model
2. Map Language Commonalities to Unified Core Syntax
3. Implement the Core Domain Models and Grammars
4. Compile and Validate Grammars

The following sections detail these steps and the resultant initial capabilities of the "ULARCH" and "UZed" parsers. The fully operational versions are described in Section 5.5.

5.4.1 Framework for the Unified Core Domain Model. The common object classes identified during the AST analysis in Section 5.2.1 formed the basis for the unified domain model. In order to formally relate these commonalities and their intrinsic properties, an encapsulating framework was defined. Comparing both LARCH and *Z* at a high level of abstraction, we can generally characterize these languages as theories, or sets of statements used to describe some problem domain (Bai). Examples of theory statements include domain models, software systems, and formal specifications. Since both LARCH and *Z*

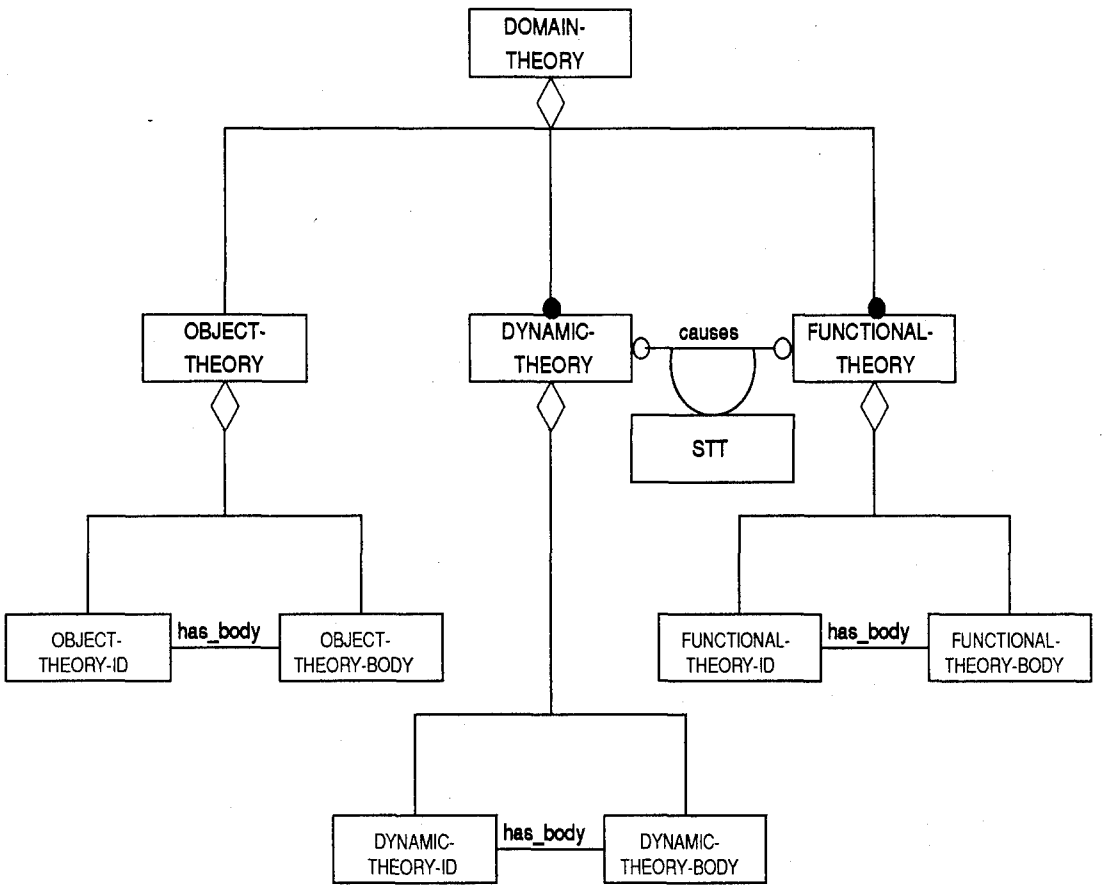


Figure 5.6 Unified Core Domain Model

define a specification as their top-level object, a corresponding root object in the unified core was defined as a "domain theory". Mapped to the Rumbaugh OOA framework, this domain theory was categorized into three distinct DomainTheoryTypes: ObjectTheory, DynamicTheory, and FunctionalTheory. In terms of multiplicity, one object theory is required, while there may be zero or more dynamic and functional theories. To complete the domain theory framework, each theory type was defined through identical aggregate components: a TheoryId and a TheoryBody. Figure 5.6 provides a graphical depiction of the core framework.

As stated above, a complete cycle of design, development, and validation activities was accomplished for each of the domain theory types. Based on the fact that every domain theory must have an object theory, the first phase concentrated on the ObjectTheory implementation. The entire process is detailed in the following three subsections. Since the other two phases for the dynamic and functional theories mirrored the first phase, their development is summarized in Subsection 5.4.5.

5.4.2 ObjectTheory Mappings. After establishing the core domain framework, the ObjectTheory commonalities of LARCH and *Z* were implemented. Starting with the layer composed of the ObjectTheoryId and ObjectTheoryBody classes, additional common object classes were identified. Since the ObjectTheoryId is a leaf node, no further decomposition was possible. The ObjectTheoryBody, however, was partitioned into two additional layers. The first tier contained the ObjectTheoryDeclarations and ObjectTheoryAxioms. The second tier, containing SignatureDeclarations and ExternalReferences, resulted from further decomposition of the theory declaration component. Figure 5.7 illustrates all the common layers within the ObjectTheory class. Below these layers, the composition of the two source languages became too divergent, thereby requiring language specific extensions. These extensions are detailed in Section 5.5.

5.4.3 Implementation of Domain Models and Grammars. The establishment of a stable domain model for the object theory facilitated the development of the corresponding REFINE code. The object classes and associations in the domain model mapped directly into REFINE'S object classes and map attributes. Analogous to the development of the original LARCH and *Z* parsers, the tree-attribute structure was defined by using object

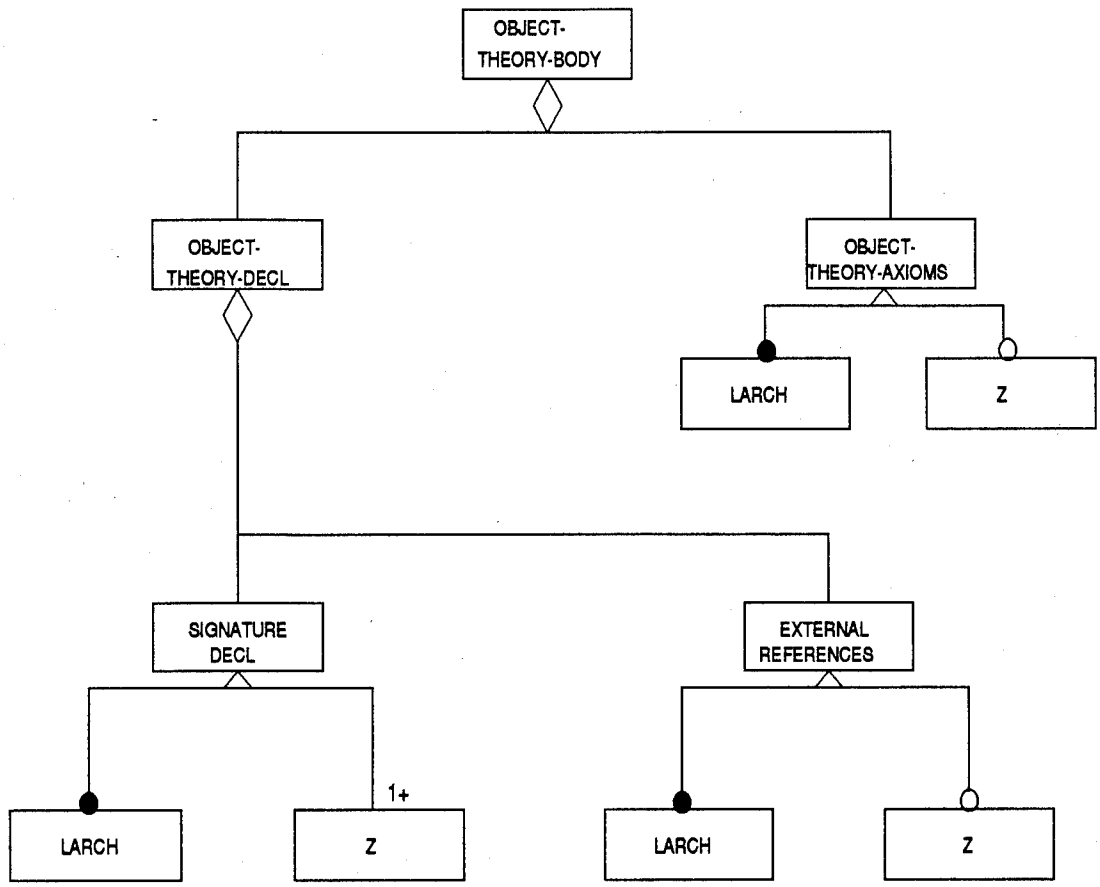


Figure 5.7 Object Theory Body Domain Model

classes for the nodes and the corresponding map attributes for the links. This tree-attribute hierarchy guided the modifications for each language's grammar. Maintaining consistency, LARCH and *Z* specific rules were carefully mapped into a target unified object theory rule. In addition to re-naming terminals and non-terminals to reflect the common unified objects, the production rules had to correctly map to each formal language's extensions.

5.4.4 Compilation and Validation of Grammars. As a direct result of each original grammar's rigorous development, the compilation of the unified grammars was straightforward. The majority of errors resulted from ill-defined attributes connecting the

unified model to the language specific models. The validation process for the unified parsers was accomplished in two phases: parsing of source programs and AST evaluations. Since this iteration of the unified parsers was focused on the object theory, just the object models of the counter and fuel tank domains introduced in Chapter III were used as inputs. A visual inspection of each domain's object theory clearly identified the boundary between the common object classes and the specialized ones. Traversing the two ASTs, each node was inspected to verify the presence of expected attribute values. As an additional confirmation, the ULARCH and UZed ASTs were compared against their respective language-specific ASTs. This process verified the preservation of each language's syntax and semantics, thus validating a common framework for parsing *Z* and LARCH specifications.

5.4.5 DynamicTheory and FunctionalTheory Iterations. As stated in Subsection 5.4.1, the iterations for both the dynamic and functional theories were symmetrical to the object theory iteration. The object classes of TheoryDeclarations, SignatureDeclarations, ExternalReferences, and TheoryAxioms were all instantiated with instances of DynamicTheory and FunctionalTheory. Complete graphical models are located in Appendix E, and Appendix F contains the corresponding REFINE implementation. For the validation of both theory types, the counter and fuel tank domains were augmented with the appropriate dynamic and functional models. The confirmation of this stage baselined the Unified Core Model's capabilities. In order to complete the functionality of these two unified parsers, the final phase required the addition of language specific extensions.

5.5 *Development of Language Specific Extensions*

Since the top-level hierarchy of both languages inherited from the common core framework, the required changes for the language extensions consisted of mapping each specific syntax to the inherited core. Reusing the language specific objects and map attributes defined in the original parsers, the only modifications consisted of re-defining the map attributes that linked the core framework with the language specific extensions. As an example, Figures 5.8 and 5.9 illustrate the ULARCH and U*Z*ed specific extensions of the inherited SignatureDeclaration class.

Although the majority of changes for both language extensions was similar, some individual syntax alterations were also necessary. Specifically, for ULARCH, LaTeX notations were required in the grammar. Since both LARCH and *Z* specifications use LaTeX notations, the production rules of the ULARCH grammar were changed to incorporate the LaTeX notation as well. Also, since there are significantly more syntactical units in *Z* than in LARCH, the U*Z*ed extension process was compounded. The following list details the major tasks involved:

1. Distinction Between State and Event Schemas - Correlated to the original parser, U*Z*ed's DynamicTheory object was further partitioned into StateTheory and EventTheory subtypes. Their internal structures were identical to their parent object, composed of declarations, external references, and axioms.

2. Inclusion of Definition Paragraphs - As detailed in Wabiszewski's thesis (Wab94), the original *Z* parser recognizes two major types of paragraphs: schema and definition. The schema paragraph types mapped directly to the unified object, dynamic, and

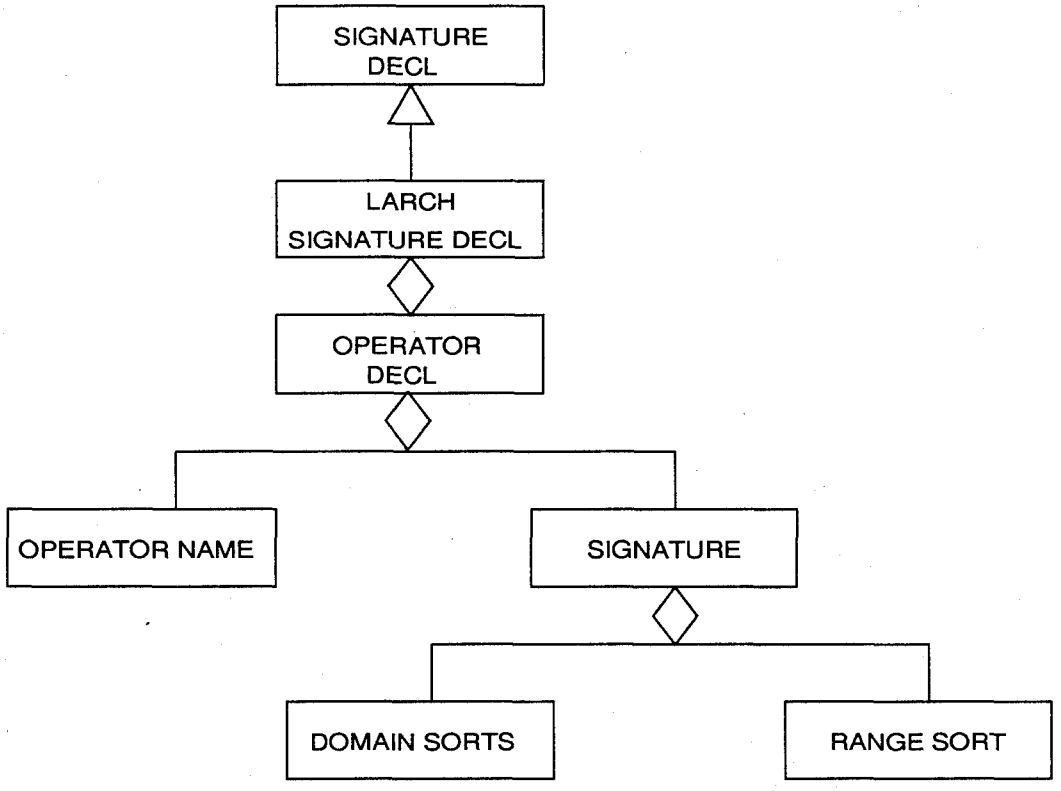


Figure 5.8 Signature Declaration - ULARCH Specific Extension

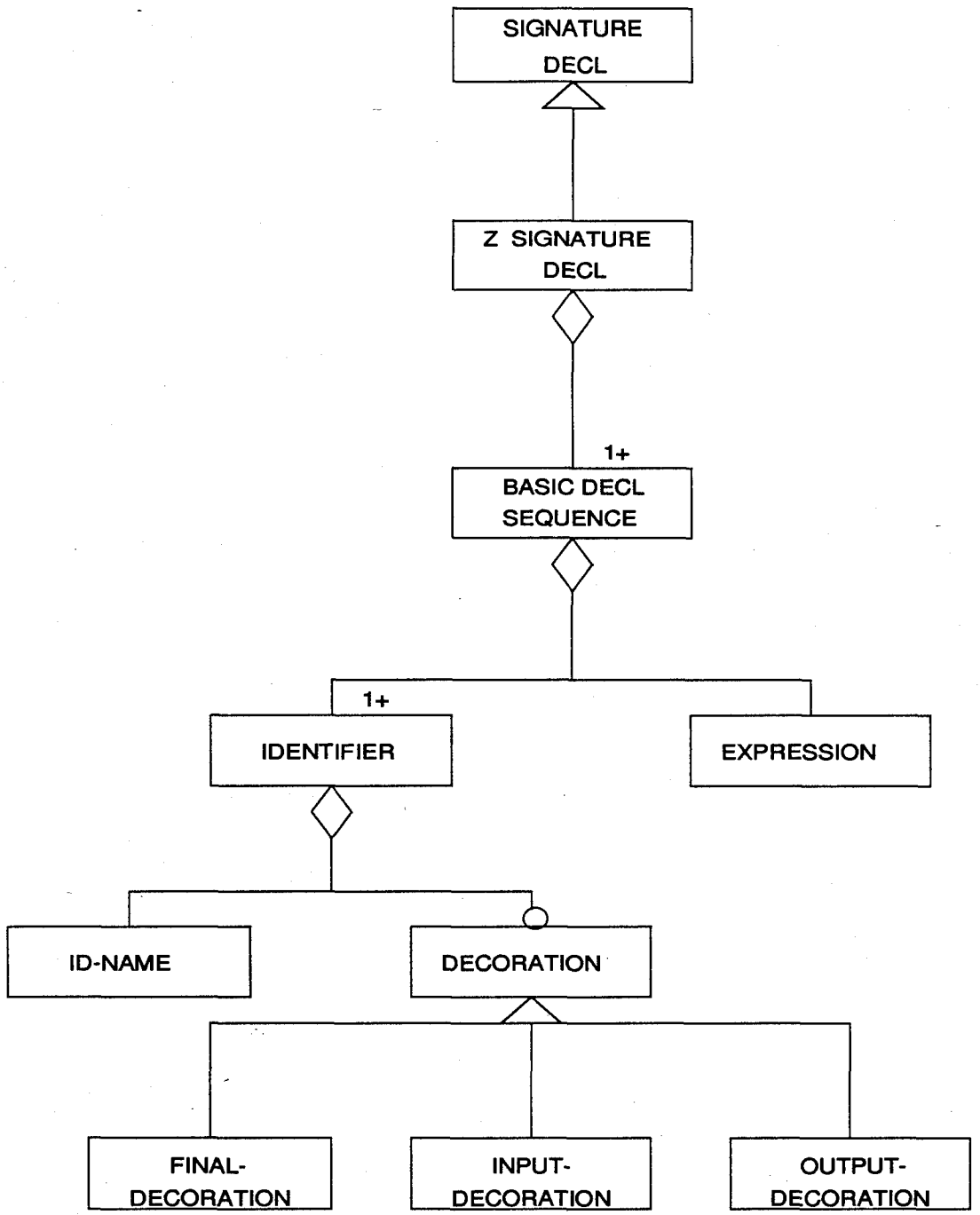


Figure 5.9 Signature Declaration - UZed Specific Extension

functional theories discussed above. The definition paragraphs were designated as another subtype of DomainTheoryType, namely a DefinitionTheory. Consequently, all subtypes of DefinitionTheory were successfully appended to the core framework.

3. Inheritance of the Mathematical ToolKit - In order to create a unified version of the ToolKit, the information pertaining to the inherited grammar was modified. Since the ToolKit is appended to the core *Z* parser at a low level, the changes to the UToolKit were minimal.

The complete set of diagrams for the ULARCH extensions are contained in Appendix G, while the REFINE code for the domain model and grammar are located in Appendices H and I, respectively. The equivalent UZed appendices can be found in Wabiszewski's thesis (Wab94).

5.6 Summary

Although the syntactic domains of the two source languages, *Z* and LARCH, were different, their semantic domains were conceptually the same. Both formal specifications encapsulate theories about a target problem domain through sets of signatures and axioms. The notion of language inheritance produced a framework for a unified core domain model and specialized language *dialects*. Intuitively, the languages' specializations should be minimal, since they are both founded on mathematical constructs; however, during this research effort, the lack of a *verifiable* mechanism for pattern matching and term rewriting limited the depth of the inherited core to a much higher level. Nonetheless, the resultant unified design does provide an initial consolidation of various formal language representations, which, in turn, promotes reusability and prevents escalation in the number of

VI. Design of a Formal Execution Framework[1]

6.1 Introduction

In Chapter V, Figure 5.4 depicted a transformation process that produces an executable program from a unified abstract structure. To maintain consistency, executable code generated from a formal specification requires a concrete target model. An execution framework is the initial step toward building this target model because it identifies the executable program's data structures and functions. The existing ULARCH and UZed parsers provide the syntactical basis for this execution framework, but additional semantic analysis is required to ensure consistency. Therefore, this chapter elaborates the steps necessary to complete the compilation analysis phase (refer to Figure 4.1) and then describes the design and development of an execution framework. The specific tasks are outlined below:

1. Semantic Analysis
2. Creation of an Execution Domain Model
3. Development of Execution Framework Mappings
4. Prototyping an Initial Executable Program

6.2 Semantic Analysis

The semantic analysis portion of the compilation process consists of various checks that ensure a program fits together in a meaningful way prior to generating an executable program (ASU88:5). For this research effort, two semantic analysis tasks were identified

[1]This chapter was co-written with Captain Kathleen Wabiszewski. It also appears in (Wab94).

to simplify the translation phase: shorthand expansion and type checking. In general, shorthand expansion augments the AST by including the signatures and axioms of any referenced components. Type checking ensures that operators used in expressions are not applied to incompatible operands. Figure 6.1 illustrates the sequencing of these two tasks.

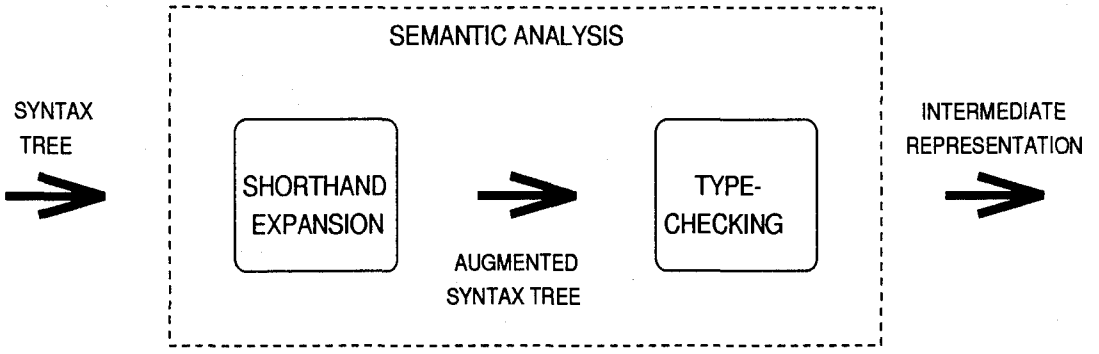


Figure 6.1 Semantic Analysis Phases

While the following subsections describe the analysis and design performed for the languages' shorthand expansion, it was decided that type checking would not be addressed during this research effort for two reasons:

1. The existence and accessibility of stand-alone type-checkers, i.e., MIT's Larch Shared Language (LSL) Checker (GH93) and DePaul University's *Z* Type Checker (ZTC) (Jia94), provided an interim verification capability.
2. Since type checking is essentially an implementation task, it could be reserved for future enhancements.

6.2.1 Shorthand Expansion Analysis. In an abstract specification, shorthand notations are used to promote modularity and encapsulation through a reduction in the amount of formal text. To translate a specification into a concrete implementation requires

the expansion of these notations to produce a robust intermediate representation. In a LARCH specification, shorthand expansion consists of unioning the operators and axioms of the "includes" trait with the original referencing trait. In a *Z* document, there can be many different types of shorthand notation, e.g., the Δ/Ξ operations and the numerous schema calculus expressions, such as schema inclusion.

In terms of the unified model, the schema inclusion notation is comparable to LARCH's "includes" mechanism, whereby the schemas' signatures are merged and the predicates are conjoined (Lig91:42). In contrast, however, the Δ and Ξ notations do not have LARCH counterparts. Described in Chapter IV of Wabiszewski's thesis (Wab94), these prefixes can be attached to a schema name in order to indicate types of schema importation. To demonstrate the underlying semantics of the two conventions, consider the following schema, S, as a reference point.

$$
\begin{array}{|l}
\hline
S \\
i, j : \mathcal{N} \\
\hline
i > j \\
\hline
\end{array}
$$

Figure 6.2 Example Schema S

The succinct notation of the Δ and Ξ conventions are used to distinguish updates from observations. Specifically, *Z*'s Δ convention, as in other areas of mathematics, is used to signify a change in state. Recalling that the tick (') decoration indicates a post-operation value, the implicit definition of this notation is given as: $\Delta State \mathrel{\widehat=} [State; State']$. The syntax of the expanded Δ schema for schema S is provided in Figure 6.3.

In Z, the Ξ symbol, chosen because of its similarity in appearance to the equivalence symbol ($\equiv$), signifies the state of the schema before and after an operation where all of the schema's components remain unchanged (Lig91:45). The default definition of the Ξ convention is as follows: $\Xi State \mathrel{\widehat=} [\Delta State \mid$ v1' = v1 $\wedge$ v2' = v2 $\wedge$ ... vk' = vk], where v1. .vk inclusive are the variables of *State*. A visual representation of an expanded Ξ schema is depicted in Figure 6.4.

ΔS

$i, j : \mathcal{N}$
$i', j' : \mathcal{N}$

$i > j$
$i' > j'$

Figure 6.3 Expanded Delta Schema

ΞS

$i, j : \mathcal{N}$
$i', j' : \mathcal{N}$

$i > j$
$i' > j'$
$i = i'$
$j = j'$

Figure 6.4 Expanded Xi Schema

To address the wide variety of shorthand notations, two alternative expansion approaches were evaluated. The first alternative combines the parsing task with the shorthand expansion, while the second alternative maintains a separation between syntax and semantics.

6.2.1.1 Addition of Explicit Production Semantics. Although the compilation analysis phase depicted in Figure 4.1 distinguished between the parsing and semantic analysis phases, some language compilers do merge these two tasks. Supporting this approach, DIALECT possesses the capability to embed semantic analysis tasks within a language's production rules. The DIALECT Users Manual outlines two situations where these explicit production semantics are effective (Sys90:5-17):

- Productions where the user must set attributes of an object that are not fully described by the syntax.
- Productions that are used to augment the syntactic inheritance hierarchy.

The second instance appeared to be applicable to this research effort; however, a prototyping effort yielded disappointing results. The production semantics, as defined in the DIALECT manual, did not appear to augment the ASTs, and it could not be discerned whether the added production semantics had any effect on the languages' rules. Furthermore, it seemed that the production semantics required terminal objects (i.e., the lowest level objects in the domain models) in its definition, which constrained the effective range of this approach. Therefore, this alternative was deemed inappropriate.

6.2.1.2 Development of Traversal Algorithms. The second alternative for augmenting the ASTs required the creation of tree traversal algorithms. Essentially, this methodology uses REFINE programs to manipulate the object base created during parsing, thereby expanding the shorthand notations. As a prototype, the following generic traversal steps were developed:

1. Locate the shorthand notation objects.

2. For each included object, collect its signatures and axioms.

3. Augment the object base with the collected signatures and axioms.

After prototyping the predefined algorithm, the resultant ASTs were viewed for accuracy and completeness. The test specification had been augmented with the collected signatures and invariants. Also, because this approach maintained disjoint syntactical and semantic phases, the grammars' production rules were not corrupted. Supported by this evidence, this alternative was selected for implementing shorthand expansions.

6.2.2 Implementation of Traversal Algorithms. For manageability, the implementation of the traversal algorithms was divided into two separate phases. Contained in Appendix J, the completed routines are depicted in the structure chart in Figure 6.5.

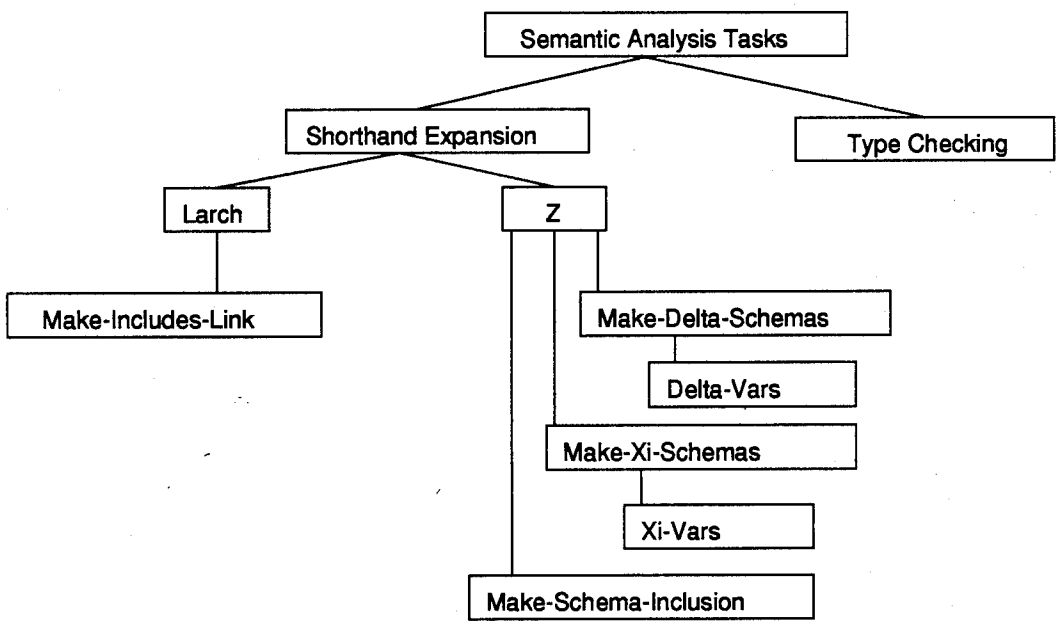


Figure 6.5 Traversal Routines

In the first implementation phase, the successful test algorithm was directly translated into routines for the ULARCH "includes" notation and UZed's schema inheritance. A summary of the respective routines is provided:

1. MAKE-INCLUDES-LINK: An "Including" trait's theory is expanded by unioning the *Included* trait's operators and axioms.
2. MAKE-SCHEMA-INCLUSION: A state schema's signature is unioned with the referenced object schema's signature. Additionally, the predicates of both schemas are conjuncted together within the state schema's predicate.

For the second phase of the iteration, based on frequency of use and correlation to the Rumbaugh model, the more complicated Δ and Ξ notations were implemented. Due to time constraints, the algorithms for the less frequently used Z schema calculus expressions were reserved for future enhancements. The specific routines developed for the two conventions are listed below:

1. MAKE-DELTA-SCHEMAS: A Δ reference is expanded by unioning the signature of the Δ schema with the signature of the referenced schema. Likewise, the predicates of both schemas are conjuncted together. A boolean annotation attribute, *delta-link*, is set to true.
    - MAKE-DELTA-VARS: If its *delta-link* attribute is true, the Δ signature is expanded a second time to include the ticked counterparts of the referenced object's variables. Also, the predicate is expanded to include the corresponding constraints placed on those ticked variables (See Figure 6.3).

2. MAKE-XI-SCHEMAS: A Ξ reference is expanded by unioning the signature of the Ξ schema with the signature of the referenced schema. Likewise, the predicates of both schemas are conjuncted together. A boolean annotation attribute, *xi-link*, is set to true.

    - MAKE-XI-VARS: If its *xi-link* attribute is true, the Ξ signature is expanded a second time to include the ticked counterparts of the referenced object's variables. Also, the predicate is expanded to include the corresponding constraints placed on both the ticked and unticked variables (See Figure 6.4).

6.2.3 Validation of Algorithms. The testing and validation of the traversal algorithms consisted of visually inspecting the augmented ASTs with OBJECT INSPECTOR. During this process, two types of errors were detected: misplacement within the hierarchy and replication of variables. In the first case, incorrect attribute references caused the "included" signatures and axioms to be appended to the wrong parent class. In the second case, the temporary set used to collect the referenced signatures and axioms was not being emptied between enumerations. Upon correction of these errors, the traversal algorithms generated the correct results, thereby producing robust intermediate representations of the source languages. The next phase of development focused on the target execution framework.

6.3 Creation of an Execution Domain Model

After completing the compilation analysis phase, the first step in designing an execution framework required the creation of a domain model for the target execution language,

REFINE. Analyzing the syntax of the language produced four major object classes as the target data structures: object classes, map attributes, functions, and rules. The object classes and map attributes were identified as terminal objects, while the function and rule classes each decomposed into local parameters and a set of preconditions and postconditions. Figure 6.6 illustrates the major object classes of the execution domain model. Once defined, the objects and associations in the domain model mapped directly into the object classes and map attributes of a REFINE description. This description reflected the execution domain model, thus establishing a concrete target for the execution framework mappings. (See Appendix K for the complete graphical model and Appendix L for the corresponding REFINE implementation.)

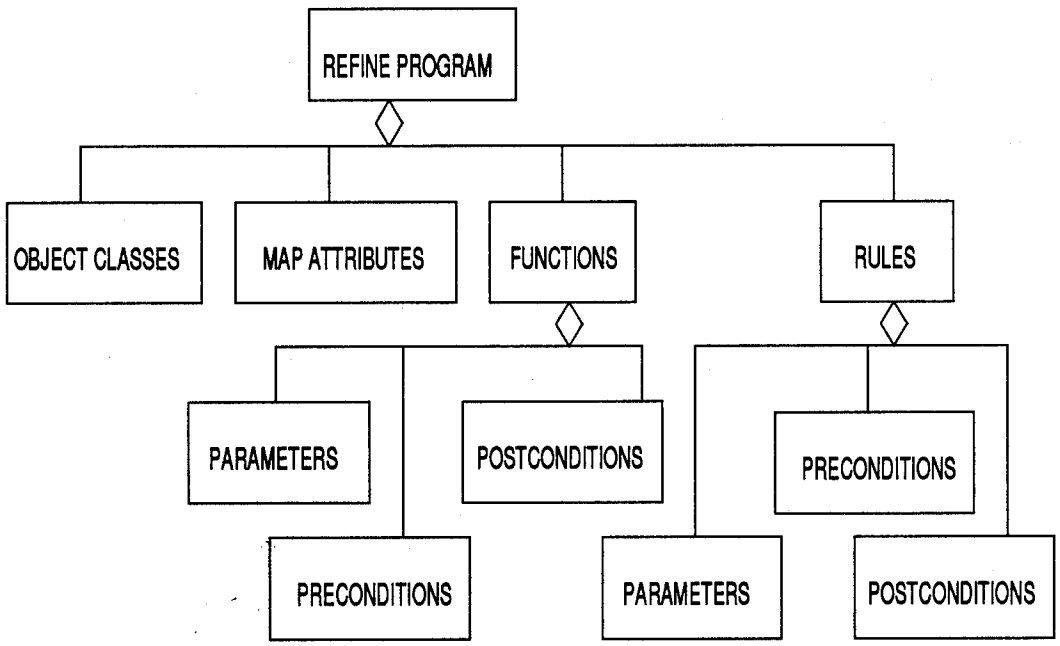


Figure 6.6 Execution Target Domain Model

6.4 Development of Execution Framework Mappings

Developing a translation process between the source and target domain models required mapping the ULARCH and UZed objects to the corresponding REFINE objects. Three execution maps were developed to correlate with the unified framework: an object theory, a dynamic theory, and a functional theory map. A two phase process was used for each mapping. First, an initial translation model defined the appropriate REFINE object classes. Since an object theory models the structure of a system and may identify some initialization routines, the target objects for this mapping were defined as REFINE object classes, map attributes, and functions. REFINE functions were also defined for both the dynamic and functional theories, since the former captures a system's static states and events, while the latter models data transformations.

The second phase of each mapping process required an analysis of both the ULARCH and UZed models in order to match their individual structures to the corresponding REFINE target components. While similar dynamic and functional theory mappings were produced, the object theory mappings were significantly different because UZed's object theory predicates are realized after state initialization, i.e., within the subsequent dynamic and functional theories. As a result of this difference, the lower right portion of Table 6.1 remains empty, while Table 6.2 is complete.

6.5 Prototyping an Initial Executable Program

The feasibility of generating code from the execution maps was demonstrated by prototyping an executable shell for the counter and fuel tank specifications. The execution shells contained the required object classes, map attributes, and function names for

Table 6.1 ULARCH and UZed Object Theory Maps

Execution Target Object	ULARCH Source Object	UZed Source Object
Object Class Name	Theory Id	Object Theory Id
Map Attributes	Tuple Object	Object Theory Declarations
Attribute Name	Tuple Field Ids	Basic Identifiers
Attribute Domain	Theory Id	Object Theory Id
Attribute Range	Tuple Field Sort Id	Any Expressions
Functions	Operators	
Function Name	Operator Name	
Function Parameters	Quantifier Variable Ids	
Function Parameter Type	Quantifier Sort Id	
Function Return Type	Operator Range Id	
Function Let Statements	Theory Axioms	
Function PreConditions	Theory Axioms	
Function PostConditions	Theory Axioms	

creating an executable program. The shells were created by a sequence of four translation algorithms, illustrated in Figure 6.7 and described below:

1. *Make Object Class:* Translates application objects in a formal specification into REFINE object classes.

Table 6.2 ULARCH and UZed Dynamic and Functional Theory Maps

Execution Target Object	ULARCH Source Object	UZed Source Object
Functions	Operators	State/Event/Fun Theory Body
Function Name	Operator Name	State/Event/Fun Id
Function Parameters	Quantifier Variable Ids	State/Event/Fun Declarations
Function Parameter Type	Quantifier Sort Id	Any Expressions
Function Return Type	Operator Range Id	
Function Let Statements	Theory Axioms	Identifiers
Function PreConditions	Theory Axioms	Any State/Event/Fun Predicates
Function PostConditions	Theory Axioms	State/Fun Post Predicates

2. *Make Attributes:* Finds the formal representation of an object's attributes and translates them into map attributes with the appropriate domain and range values.

3. *Make Functions:* Translates the formal operations defined in ULARCH and UZed and creates a corresponding function name.

4. *Print Program:* Outputs the executable shell. The appropriate object classes, map attributes, and function names are listed in the correct REFINE format.

Appendix M contains the initial translation algorithms.

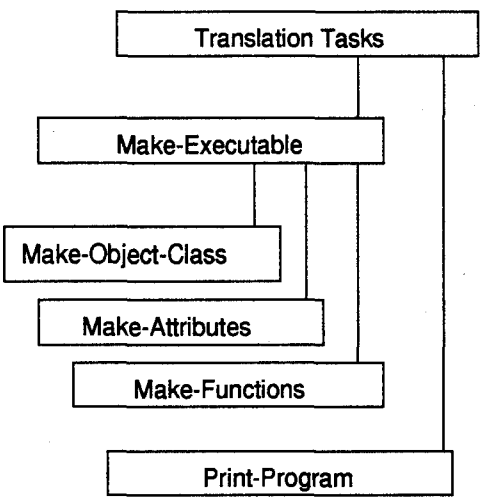


Figure 6.7 Translation Algorithms

6.5.1 Deficiencies in the Execution Maps. Constrained by time, the execution mappings were limited to a subset of the major object classes within the REFINE target domain model (refer to Figure 6.6). As a result, the initial prototyping effort did not complete the translation mechanism for the target model's function and rule object classes. These uncorrected deficiencies are detailed below.

1. *Make-Functions:* This mapping does not currently translate formal parameters and theory axioms into appropriate function parameters, preconditions, and postconditions. The translation must differentiate between theory axioms used as preconditions and those used as postconditions.

2. *Make-Rules:* The algorithm and mapping for this REFINE object class does not exist. This translation determines the sequences of events based on the state transition tables.

6.6 Summary

Developing an execution framework proved extremely useful for demonstrating the feasibility of generating code from a formal specification. The framework established the target data structures and functions for a consistent and correct translation process. It also became the foundation for developing the execution maps from the unified source models into the REFINE target model. From the execution maps, specific translation algorithms were created to generate executable REFINE code for the ULARCH and UZed counter and fuel tank specifications.

VII. Conclusions and Recommendations

This chapter summarizes the accomplishments achieved in this research and describes the general and specific conclusions concerning the development of a formal object transformation process. Additionally, this chapter presents some recommendations for future research.

7.1 Summary of Accomplishments

The objective of this study was to formalize the object-based composition approach using LARCH. Three steps were defined to accomplish this objective:

1. Design and develop an algebraic transformation process using OMT models as the source representation and LARCH as the target representation
2. Develop a unified abstract framework by comparing the similarities and differences between LARCH and *Z*.
3. Based on the unified abstract structure, design an initial execution framework

These steps were accomplished using an evolutionary development approach. The first evolution produced an algebraic transformation process that formalizes the OMT object, dynamic, and functional models. This evolution demonstrated that LARCH contains suitable notations for expressing object-oriented concepts. The second and third evolutions produced a robust, unified LARCH parser demonstrating the notion of language inheritance. Additionally, the unified LARCH parser established a stable foundation for developing an execution framework, the objective of the last evolution. Together, these

evolutions demonstrate the feasibility of formally extending object-oriented domain models using algebras and translating the formal models into an execution framework.

7.2 General Conclusions[1]

Listed below are general conclusions drawn from this research:

1. Analyzing the abstract syntax trees of LARCH and *Z* specifications demonstrated that these formal languages contain a common set of core constructs suitable for creating a canonical framework for formalized object models. This unified framework fulfills the notion of language inheritance whereby a a general model for formal languages exist with small variants known as "dialects". Additionally, the development of a unified model establishes a front-end for formal system composition. This front-end can support three principal processes: code generation, theorem proving, and design refinement, as depicted in Figure 7.1. The unified model forms the basis for creating interface languages for the purpose of specification execution. Theorem proving sentences can also be generated from the unified model for input into a theorem prover to verify a specification. Finally, the unified framework can produce synthesis diagrams to support the notion of theory-based design refinement (Smi90) (BFG+94).
2. The evolutionary development strategy is well-suited for developing a formal language compiler because the compilation phase can be decomposed into several stages. Each

[1]This section was co-written with Captain Kathleen Wabiszewski. It also appears in (Wab94).

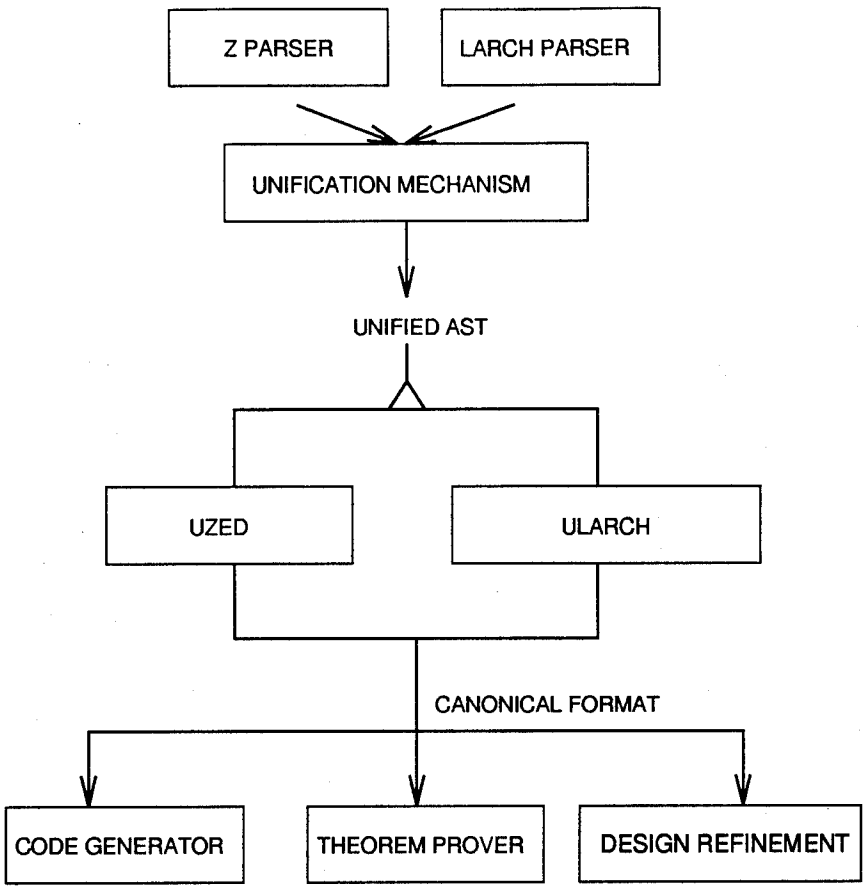


Figure 7.1 Transformation Process

stage produces an operational product demonstrating a minimum set of capabilities, and each incremental product is easily extensible.

3. Developing object-oriented domain models for LARCH and *Z* clarified the languages' structure by identifying the important objects and associations. The established domain models provided a well-defined blueprint guiding the implementation of the LARCH and *Z* grammars in REFINE.

4. The REFINE environment simplifies the development of a formal language parser and an initial execution framework by providing compilation tools, i.e., DIALECT and

OBJECT INSPECTOR. DIALECT handles token formation and creates an explicit hierarchy for LARCH using a BNF notation. These hierarchies, in the form of abstract syntax trees (ASTs), are stored in the object base and viewed using OBJECT INSPECTOR. A visual representation is extremely beneficial because it allows one to inspect the parse tree and understand the underlying structure of the formal language. ASTs also provide a strong foundation to perform semantic analysis and to begin building an execution framework.

7.3 Specific Conclusions

Specific conclusions that can be drawn from developing an algebraic transformation process and a unified LARCH parser are presented below:

1. Algebraic specifications, specifically LARCH, can formally represent object-oriented constructs. Chapter III showed how LARCH contained suitable notations to represent object attributes, inheritance, aggregation, and operator behavior.
2. The Unified LARCH Parser successfully parses a wide range of LARCH traits. Appendix N contains a user's manual for running the parser and Appendix O contains a list of successfully parsed traits.
3. The Unified LARCH Parser generates well-defined abstract syntax trees simplifying the following compilation phases:
   - Semantic Analysis
   - Code Generation

As shown in Chapter IV, the ambiguity encountered in the original grammar produced a parser that generated incredibly deep and unintelligible ASTs. Any attempt to perform semantic analysis or code generation on this structure would have caused numerous work-arounds to properly augment the tree and translate it into an executable program. On the other hand, the revised parser, based on a stable domain model, generated succinct AST representations. These ASTs simplified the semantic analysis tasks of shorthand expansion and the initial execution translation described in Chapter VI.

4. The Unified LARCH Parser is extensible. LARCH'S domain model and syntax allow developers to extend the current language model in a straightforward fashion by defining new subclasses. This flexibility was demonstrated in Chapter IV when new axiomatic terms were added to the existing domain model and grammar.

7.4 Recommendations for Future Research

Listed below are areas that require further research:

1. *Extend the Unified Abstract Framework* - The goal for establishing a unified abstract framework is to provide a common base language with small variant specializations or *dialects*. However, the current framework does not reflect a true common core because each language has not been reduced into its most significant form. Decreasing the language specific dependencies requires *pushing* the specializations down to the lowest level in the unified model. *Pushing* is defined as the consolidation and resolution of similar constructs via pattern matching and term rewriting. Pattern matching

is applied to the syntactic components of the ASTs to identify language commonalities. Mathematically verifiable procedures known as term rewriting are then used to translate the commonalities into a target canonical form. When the terms can no longer be rewritten, they are considered to be in a normal form. This form represents the complete common base language expressing the language's meaning without any loss of generality.

2. *Identify Theorem Proving Tasks* - The refinement process from a formal specification into an executable implementation should demonstrate that the specification remains consistent and complete (NS91). Proving consistency and completeness requires identifying theorem proving tasks. These tasks ensure that the implementation possesses the right properties, for example:

   - Does the specified operation function correctly?
   - Are the proper constraints on operations implemented?
   - Do the specified invariants hold over time?
   - Does the operation guarantee the correct final state based on the initial state and event?
   - Is an object prevented from being in two different states at one time?
   - Is an operation guarded by the proper pre-conditions?

3. *Complete the Execution Framework* - To complete the execution framework the following tasks are required:

- *Type Checking* - Key component during semantic analysis ensuring that operators have compatible operands (ASU88). Although a stand-alone LSL checker exists, incorporating a type checker in the REFINE environment provides a seamless compilation path for the formal object transformation process.

- *Additional Semantic Checks* - Required to complete the translation of axiomatic equations in LARCH and schema predicates in *Z* into REFINE functions. The main emphasis for this task centers around identifying the appropriate pre- and postconditions to establish the corresponding REFINE transform operations.

- *Execution Translation* - Final task in completing the execution framework. This task produces an executable program of a formally specified problem.

- *System Test* - Performed to validate the formal execution phase. The established validation domains, counter and fuel tank should be used to demonstrate and validate the specification's behavior. Additionally, larger domains should be evaluated to validate the properties of scalability and completeness.

- *System Verification* - Required to verify the consistency of the execution translation (i.e., LARCH Behavior => REFINE Behavior). For example, theorem proving tasks need to be identified to check that quantified expressions in LARCH are consistent with the quantified expressions in the REFINE program.

The recommendations described above are depicted in Figure 7.2 showing the integration of additional semantic analysis, theorem proving, and execution tasks to augment the current transformation process.

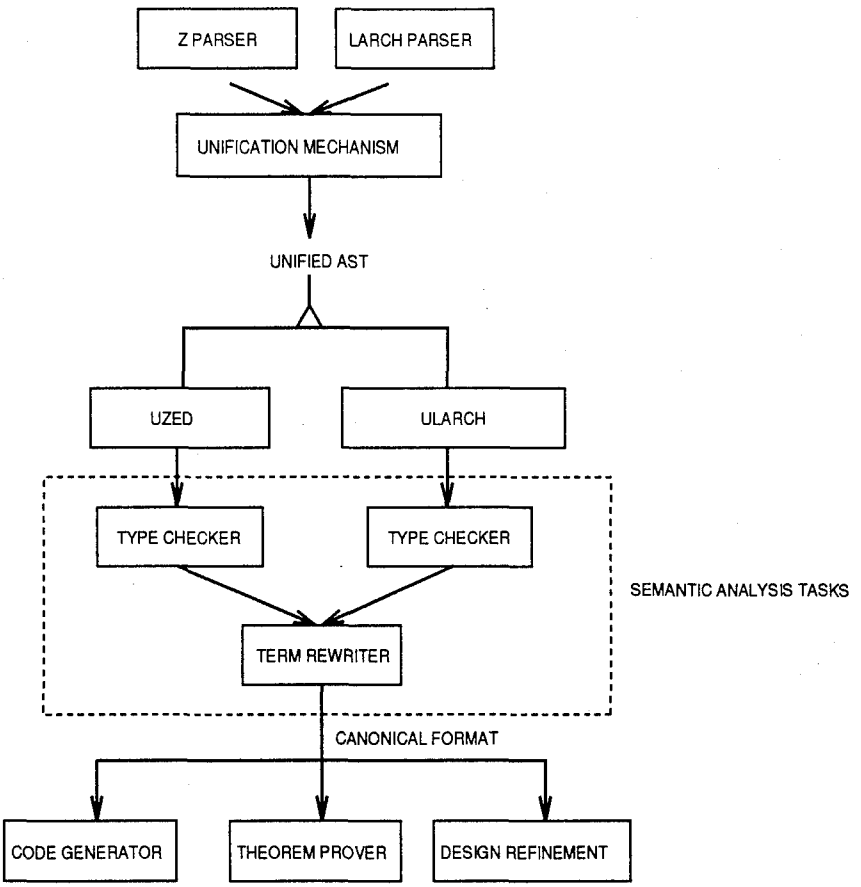


Figure 7.2 Modified Transformation Process

4. *Expand the Validation Domains* - A simplistic validation was conducted on the algebraic transformation process using the counter and fuel tank domains. To provide a robust validation of the completeness and consistency claims, additional domains need to be evaluated. One domain to consider is an Air Force Wing command and control domain designed by Hunt and Sarchet (Hun94) (Sar94).

5. *Model State Transition Tables Algebraically* - To complete the algebraic transformation process, the state transition tables should be formally represented in a LARCH specification. A formal description of a state transition table should include an ob-

ject's current state, event, and next state. In addition, the state transition description should remain consistent with the overall system description.

6. *Include Multiple Inheritance in the Algebraic Framework* - Multiple inheritance is an important object-oriented construct, but it complicates a formal specification by introducing additional operators and invariants from various "parent" traits. If a child trait contains multiple inheritance, its formal representation must use and correctly combine its parent's operators and axioms. Future enhancements to the algebraic transformation phase should consider incorporating multiple inheritance.

7. *Enhance the Unified* LARCH *Parser* - The current implementation of the LARCH parser uses parentheses to disambiguate between certain algebraic terms. These parentheses are required to provide a more concrete description of the grammar (i.e., reduce the level of abstraction), but they also force a user to a specific format for representing their traits. Therefore, future research should investigate reducing the parenthesis dependency while keeping the grammar intact.

7.5 Final Comments

Formal methods, specifically algebraic languages, play an important role in improving the specification and composition of software systems. These methods are precise and unambiguous and can complement the current object-oriented perspective by providing a seamless transition from a visual system description into a concise system specification. The algebraic-based execution framework developed during this study demonstrates the feasibility of integrating the practical object-oriented approach with the more precise algebraic-based method. In addition, this research provides an initial capability for ex-

ecuting LARCH specifications establishing a foundation for verifying the correctness of a specification prior to design and implementation. This capability improves the reliability and outcome of a software system ensuring that the simulated behavior mirrors the original specification.

Appendix A. Counter Domain Traits

This appendix contains the Counter domain's formal specifications. Developed by following the LARCH transformation heuristics described in Chapter III, these traits support the notion of producing algebraically-based system specifications. Additionally, these traits supported the validation of the unification and execution phases acting as the source (i.e., the input) to generate an executable shell. These traits have been typed-checked using the LSL checker obtained from MIT.

```
%ObjectTheory
Counter: trait
  includes Integer
  C tuple of value, limit: Integer
  introduces
    new: → C
  asserts  ∀ c: C
    new.value == 0;
    new.limit == 100;
    c.value ≥ 0;
    c.limit ≥ c.value

%FunctionalTheory
Increment: trait
  includes Counter
  introduces
    add1: C → C
  asserts  ∀ c: C
    add1(new) == [(c.value + 1), c.limit];
    add1(c) == if c.value < c.limit then [(c.value + 1), c.limit] else
                                          [c.value, c.limit]

%FunctionalTheory
Add: trait
  includes Counter
  introduces
    addvalue: C, Integer → C
  asserts  ∀ c: C, v: Int
    addvalue(new, v) == if new.limit ≥ v then [(c.value + v), c.limit] else
                          [c.value, c.limit];
    addvalue(c, v) == if (c.value + v) ≤ c.limit then [(c.value + v), c.limit] else
                        [c.value, c.limit]
```

```
%FunctionalTheory
Subtract: trait
  includes Counter
  introduces
    subvalue: C, Integer → C
  asserts  ∀ c: C, v: Int
    subvalue(c, v) == if (c.value - v) ≥ 0 then
                      [(c.value - v), c.limit] else [c.value, c.limit]
```

Appendix B. Fuel Tank OMT Analysis Models and State Transition Table

The Fuel Tank Analysis Models were selected as a validation domain for the formal object transformation process. This appendix contains the OMT analysis models and state transition table used to support the algebraic transformation process, the unification process, and the final execution process. These models were developed by Hartrum (HB94).

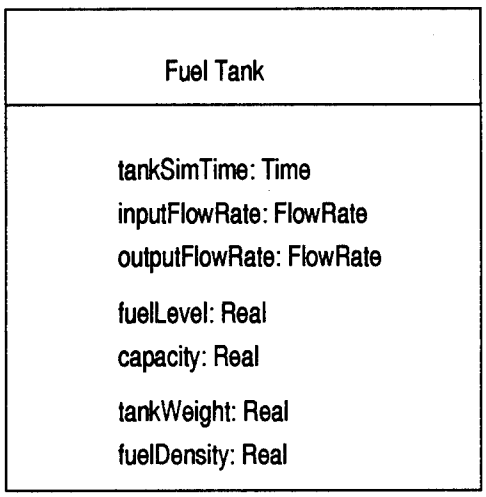


Figure B.1 Fuel Tank Object Model

Table B.1 Fuel Tank State Transition Table.

Current	Event	Guard	Next	Action
Empty	StartFill		Filling	Schedule(TankFull)
Filling	StopFill	$fuel\_level = capacity$	Full	Cancel(TankFull); update_level
Filling	StopFill	$fuel\_level < capacity$	PartiallyFilled	Cancel(TankFull); update_level
Filling	TankFull		Full	Overflow; update_level
Filling	StartUse		FillAndUse	Cancel(TankFull); update_level
Full	StartFill		Full	Overflow
Full	StartUse		Using	Schedule(TankEmpty)
Using	TankEmpty		Empty	ChangeFuelFlow(0); update_level
Using	StopUse		PartiallyFilled	Cancel(TankEmpty); update_level
Using	StartFill		FillAndUse	Cancel(TankEmpty); update_level
PartiallyFilled	StartFill		Filling	Schedule(TankFull)
PartiallyFilled	StartUse		Using	Schedule(TankEmpty)
FillAndUse	StopUse		Filling	Schedule(TankFull); update_level
FillAndUse	StopFill		Using	Schedule(TankEmpty); update_level

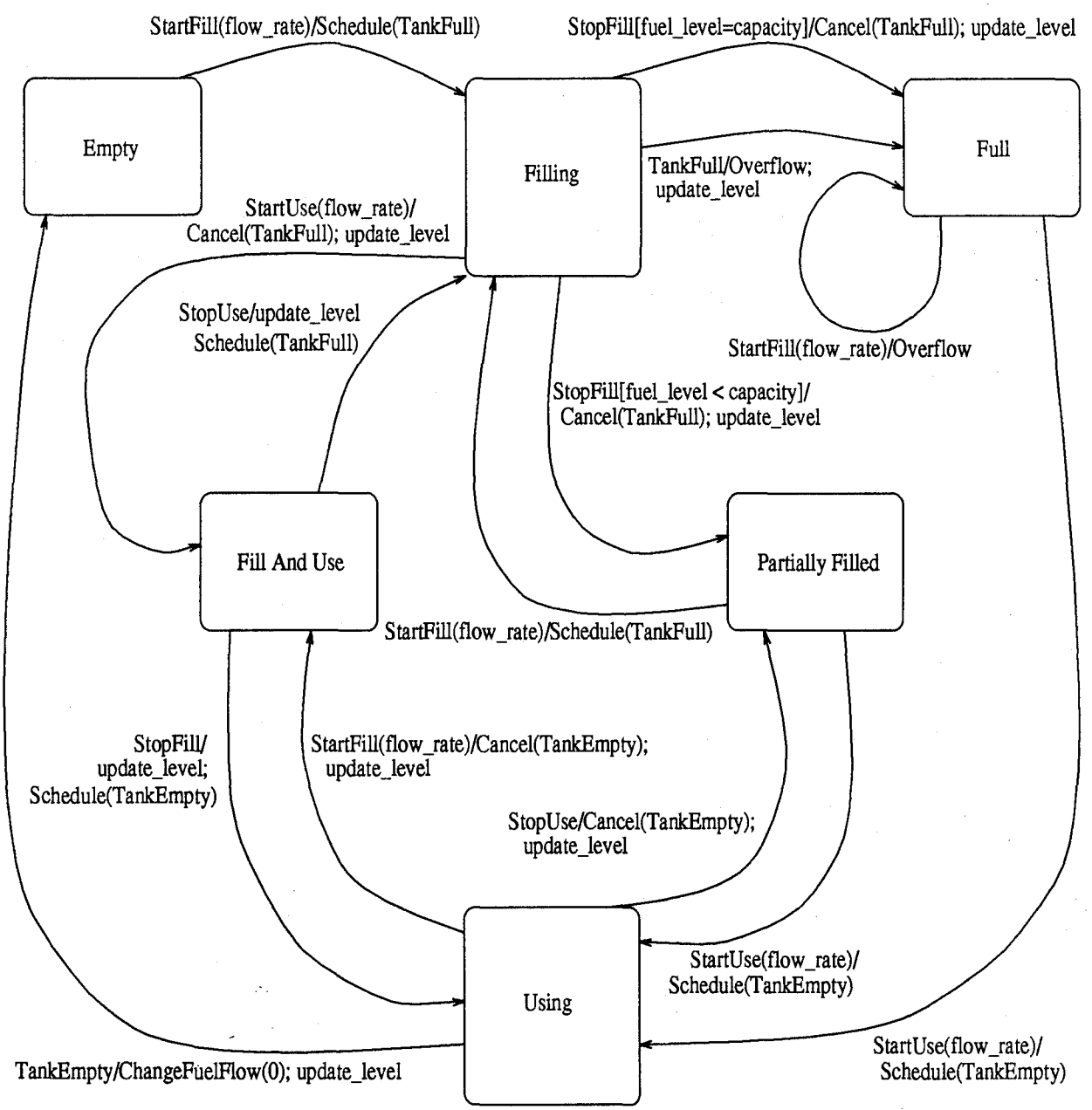


Figure B.2 Fuel Tank Dynamic Model

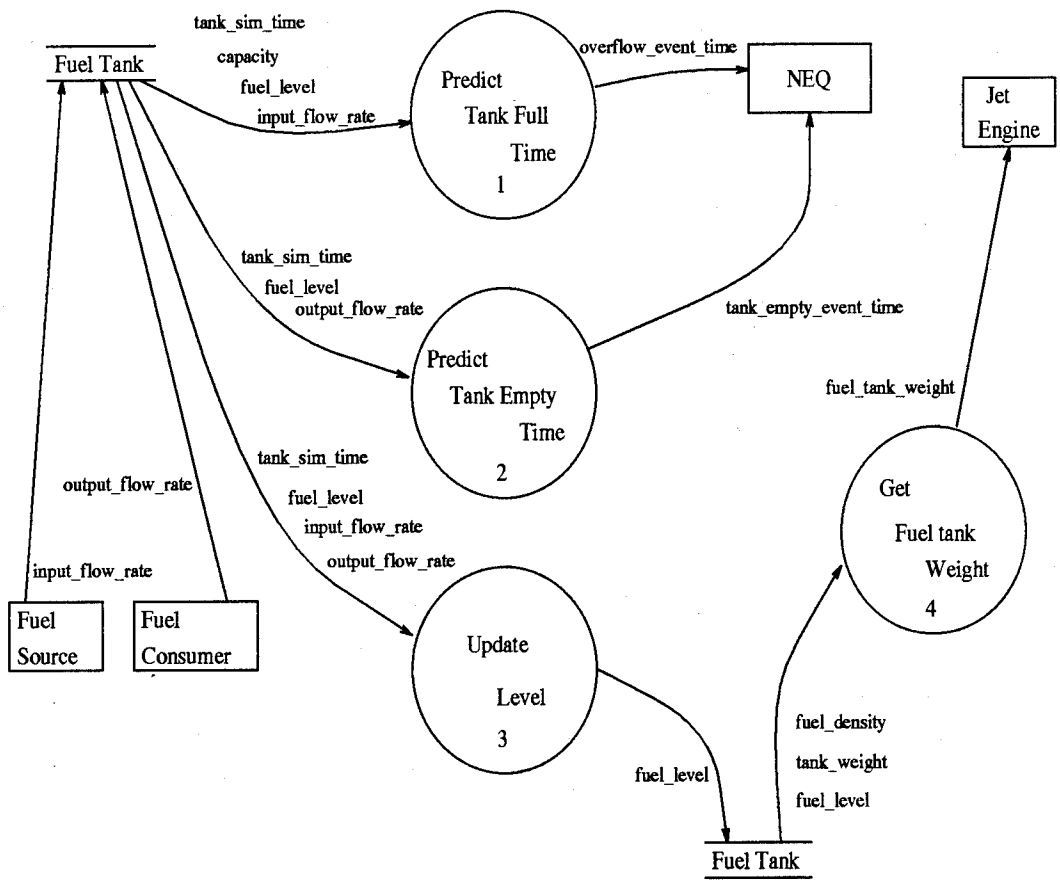


Figure B.3 Fuel Tank Functional Model - 1

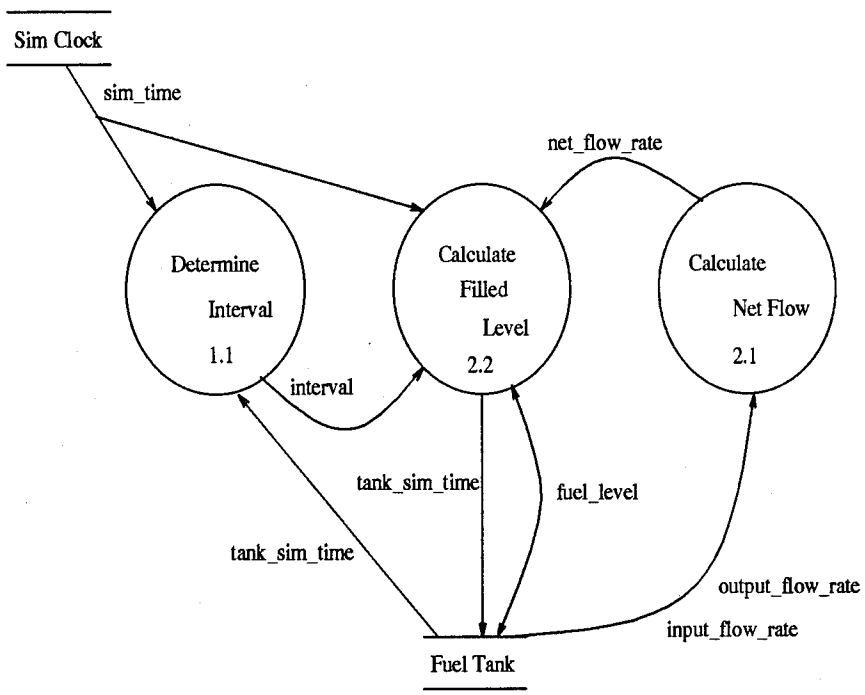


Figure B.4 Fuel Tank Functional Model - 2

Appendix C. Fuel Tank Domain Traits

This appendix contains the Fuel Tank domain's formal specifications. Developed by following the algebraic transformation heuristics outlined in Chapter III, these traits establish the feasibility of formally extending the OMT analysis models using LARCH. These traits were used as input into the unification and execution phases to produce an initial executable Fuel Tank program. All of the traits have been typed-checked using the LSL checker obtained from MIT.

```
%ObjectTheory
FuelTank: trait
  includes Integer
  FT tuple of tankSimTime: T, inputFlowRate, outputFlowRate: FR,
              fuelLevel, capacity, tankWeight, fuelDensity: Int
  introduces
    fuelTankWeight: FT → Int
  asserts  ∀ ft: FT
    ft.fuelLevel ≤ ft.capacity;
    fuelTankWeight(ft) == ft.tankWeight + (ft.fuelDensity * ft.fuelLevel)

%DynamicTheory
EmptyState: trait
  includes FuelTank(Int for FR)
  introduces
    empty: FT → Bool
  asserts  ∀ ft: FT
    empty(ft) == (ft.fuelLevel = 0) ∧ ((ft.inputFlowRate = 0) ∧ (ft.outputFlowRate = 0))

%DynamicTheory
FullState: trait
  includes FuelTank(Int for FR)
  introduces
    full: FT → Bool
  asserts  ∀ ft: FT
    full(ft) == (ft.fuelLevel = ft.capacity) ∧ ((ft.inputFlowRate = 0)
                ∧ (ft.outputFlowRate = 0))

%DynamicTheory
FillingState: trait
  includes FuelTank(Int for FR)
  introduces
    filling: FT → Bool
  asserts  ∀ ft: FT
    filling(ft) == ((ft.fuelLevel ≥ 0) ∧ (ft.fuelLevel ≤ ft.capacity))
                 ∧ ((ft.inputFlowRate > 0) ∧ (ft.outputFlowRate = 0))
```

```
%DynamicTheory
UsingState: trait
  includes FuelTank(Int for FR)
  introduces
    using: FT → Bool
  asserts ∀ ft: FT
    using(ft) == ((ft.fuelLevel ≥ 0) ∧ (ft.fuelLevel ≤ ft.capacity))
                 ∧ ((ft.inputFlowRate = 0) ∧ (ft.outputFlowRate > 0))

%DynamicTheory
StartFill: trait
  includes FuelTank(Int for FR)
  introduces
    signalStartFill: → Bool
  asserts ∀ ft: FT
    signalStartFill == ft.inputFlowRate > 0

%DynamicTheory
StopFill: trait
  includes FuelTank(Int for FR)
  introduces
    signalStopFill: → Bool
  asserts ∀ ft: FT
    signalStopFill == ft.inputFlowRate = 0

%DynamicTheory
StartUse: trait
  includes FuelTank(Int for FR)
  introduces
    signalStartUse: → Bool
  asserts ∀ ft: FT
    signalStartUse == ft.outputFlowRate > 0

%DynamicTheory
StopUse: trait
  includes FuelTank(Int for FR)
  introduces
    signalStopUse: → Bool
  asserts ∀ ft: FT
    signalStopUse == ft.outputFlowRate = 0

%FunctionalTheory
DetermineInterval: trait
  includes FuelTank(Int for T), SimClock(Int for T)
  introduces
    interval: FT, C → Int
  asserts ∀ ft: FT, c: C
    interval(ft, c) ≥ 0;
    interval(ft, c) == if (c.simTime > ft.tankSimTime) then (c.simTime - ft.tankSimTime)
                       else 0
```

```
%FunctionalTheory
CalculateFilledLevel: trait
  includes FuelTank(Int for FR), SimClock(Int for T), DetermineInterval
  introduces
    filledLevel: FT, Int → Int
  asserts  ∀ ft: FT, c: C, t: Int
    filledLevel(ft, t) == if (t = 0) then ft
                          else [c.simTime, ft.inputFlowRate, ft.outputFlowRate,
                                (ft.fuelLevel + (interval(ft, c) *
                                ft.inputFlowRate)), ft.capacity, ft.tankWeight,
                                ft.fuelDensity]


%FunctionalTheory
PredictTankFullTime: trait
  includes FuelTank
  introduces
    overflowEventTime: FT → T
  asserts  ∀ ft: FT, c: C, t: T
    overflowEventTime(ft) == if (ft.inputFlowRate ≠ 0)
                        then (ft.tankSimTime + ((ft.capacity - ft.fuelLevel)/
                                                ft.inputFlowRate))
                             else 0
```

Appendix D. REFINE *Code for the State Transition Table*

This appendix contains the REFINE code for the State Transition Table (STT), illustrated in Figure E.5, Appendix E. A formal domain model and grammar does not exist for the STT because of the limitations in both *Z* and LARCH to formally specify the entries in the state transition tables. Thus, to complete a formal implementation of the transformation process, a grammar and domain model were developed to parse the STT generating initial objects to be used in the execution framework.

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: stt.re

 Description: The following program defines the State Transition Table
(STT) Domain Model and the Grammar.  The STT is part of the OMT
analysis models and is required to create a complete and accurate
execution model for the formal transformation process.  Compilation of
the domain model and grammar generates an STT parser for state
transition tables created using the Rumbaugh method.  Latex specific
notations are used.

||#

% ----------------------------------------------------------------------
%  STT Class Definitions
% ----------------------------------------------------------------------

var Table               : object-class subtype-of user-object
var StateTable          : object-class subtype-of Table
var StateEntry          : object-class subtype-of Table
var Identifier          : object-class subtype-of Table

%-----------------------------------------------------------------
% STT Attribute Declarations
%-----------------------------------------------------------------

var table-name          : map(Table, Identifier) = {||}
var table-label         : map(Table, Identifier) = {||}
var state-entries       : map(Table, seq(StateEntry)) = {||}
var current-state       : map(Table, Identifier) = {||}
var state-event         : map(Table, Identifier) = {||}
var state-parameter     : map(Table, seq(Identifier)) = {||}
var state-guard         : map(Table, Identifier) = {||}
var next-state          : map(Table, Identifier) = {||}
```

```
var state-action        : map(Table, Identifier) = {||}
var num-real            : map(Table, real) = {||}
var num-int             : map(Table, integer) = {||}

%-------------------------------------------------------------
%  Structure for Abstract Syntax Tree
%-------------------------------------------------------------

form Define-Tree-Attributes-of-State-Table
  Define-Tree-Attributes('StateTable, {'table-name, 'table-label, 'state-entries})&
  Define-Tree-Attributes('StateEntry, {'current-state, 'state-event, 'state-parameter,
                                       'state-guard, 'next-state, 'state-action})


%-------------------------------------------------------------
%  STT Grammar Definition
%-------------------------------------------------------------


!!in-grammar('syntax)

grammar STT
   start-classes StateTable
   file-classes StateTable
   no-patterns

   case-sensitive
   symbol-continue-chars
   "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789._-=\\$()<>"
   symbol-start-chars
   "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_$("

  productions

   StateTable   ::= ["\\begin{table}[htb]"
     "\\caption{" table-name "}"
     "\\vspace*{.2in}"
     "\\label{" table-label "}"
     "\\centering"
     "\\begin{tabular}{|l|l|l|l||l|l|}"
     "\\hline"
     "Current" "&" "Event" "&" "Parameters" "&" "Guard" "&"
                  "Next" "&" "Action\\\\"
     "\\hline\\hline"
     state-entries + ""
     "\\hline\\hline"
     "\\end{tabular}"
     "\\end{table}"] builds StateTable,

   StateEntry   ::= [current-state "&" state-event "&" state-parameter * ""
                     "&" {state-guard} "&" next-state "&" {state-action}
```

```
    "\\\\" {"\\hline"}] builds StateEntry,

  Identifier   ::= [(name | num-int | num-real)] builds identifier

end
```

Appendix E. Unified Domain Model

This appendix contains the object-oriented domain models for the unified abstract framework. These models identify the common object classes shared by LARCH and *Z*. The common object classes promote a uniform interface for formalizing object-oriented analysis models and are the foundation for establishing a canonical representation for formal specification languages. Each language's "dialect" (i.e., the variances of the language from the common core) is depicted in the figures as specialized object classes.

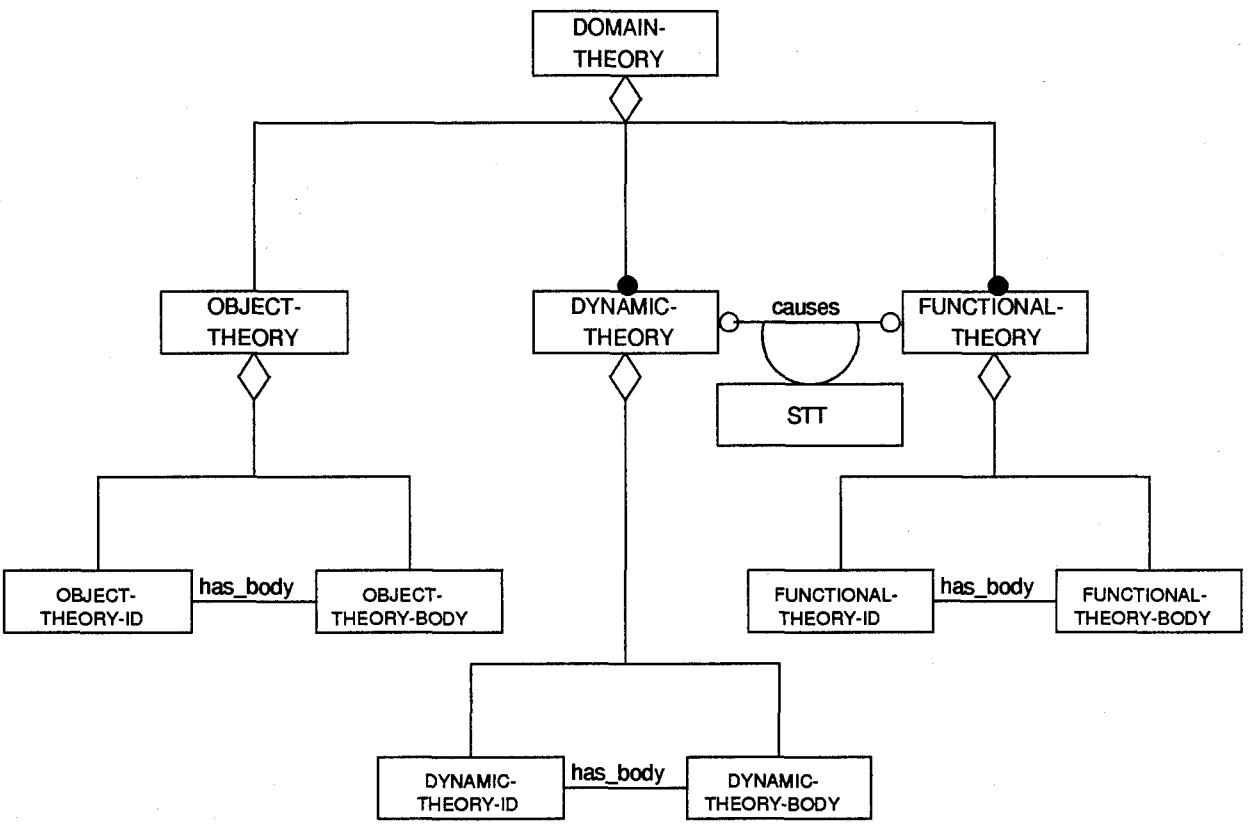


Figure E.1 Unified Domain Theory Model

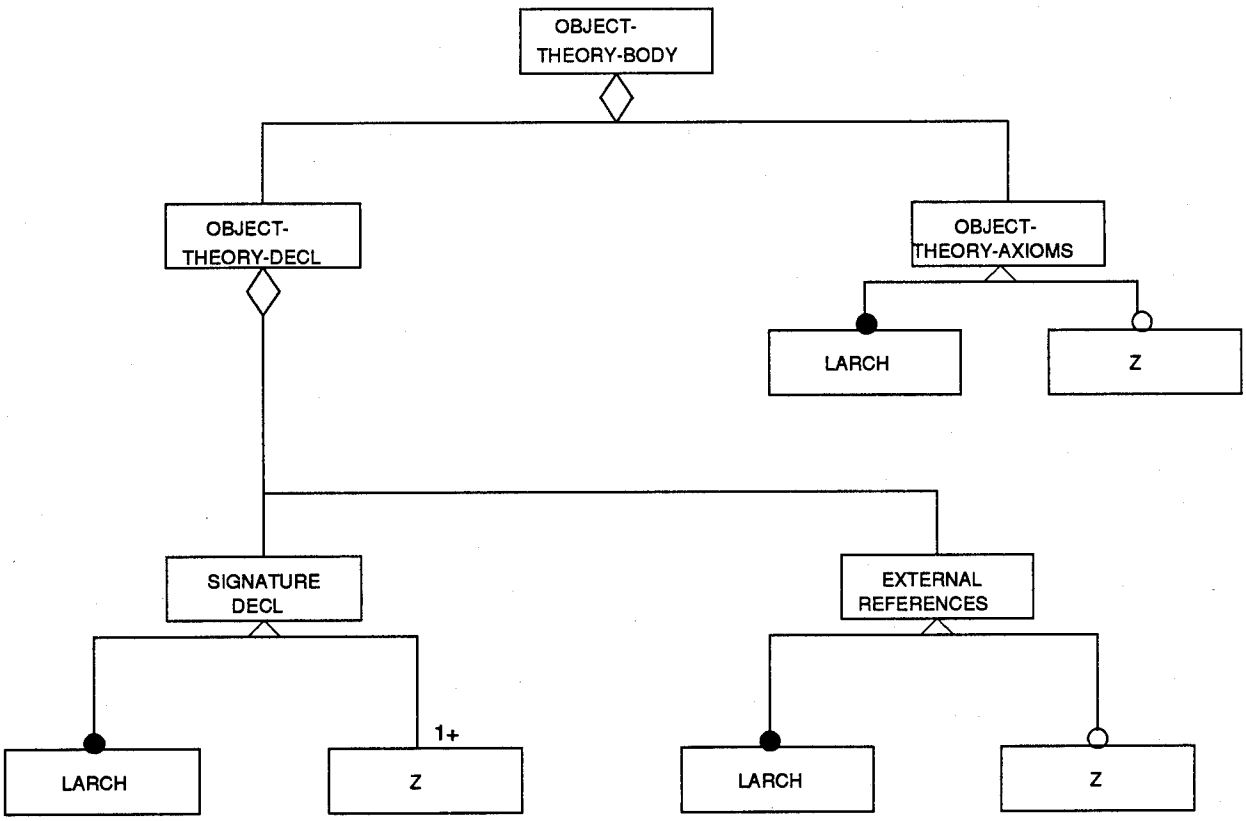


Figure E.2 Unified Object Theory Body

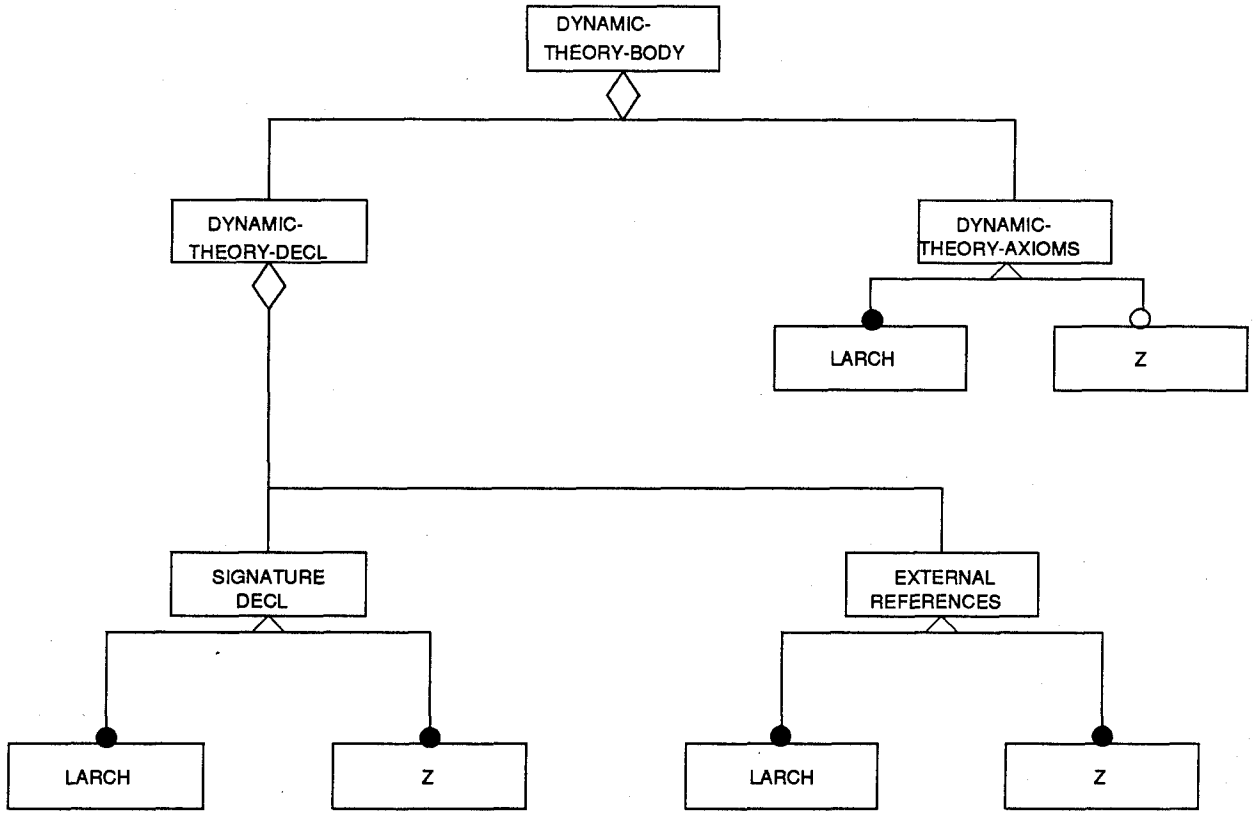


Figure E.3 Unified Dynamic Theory Body

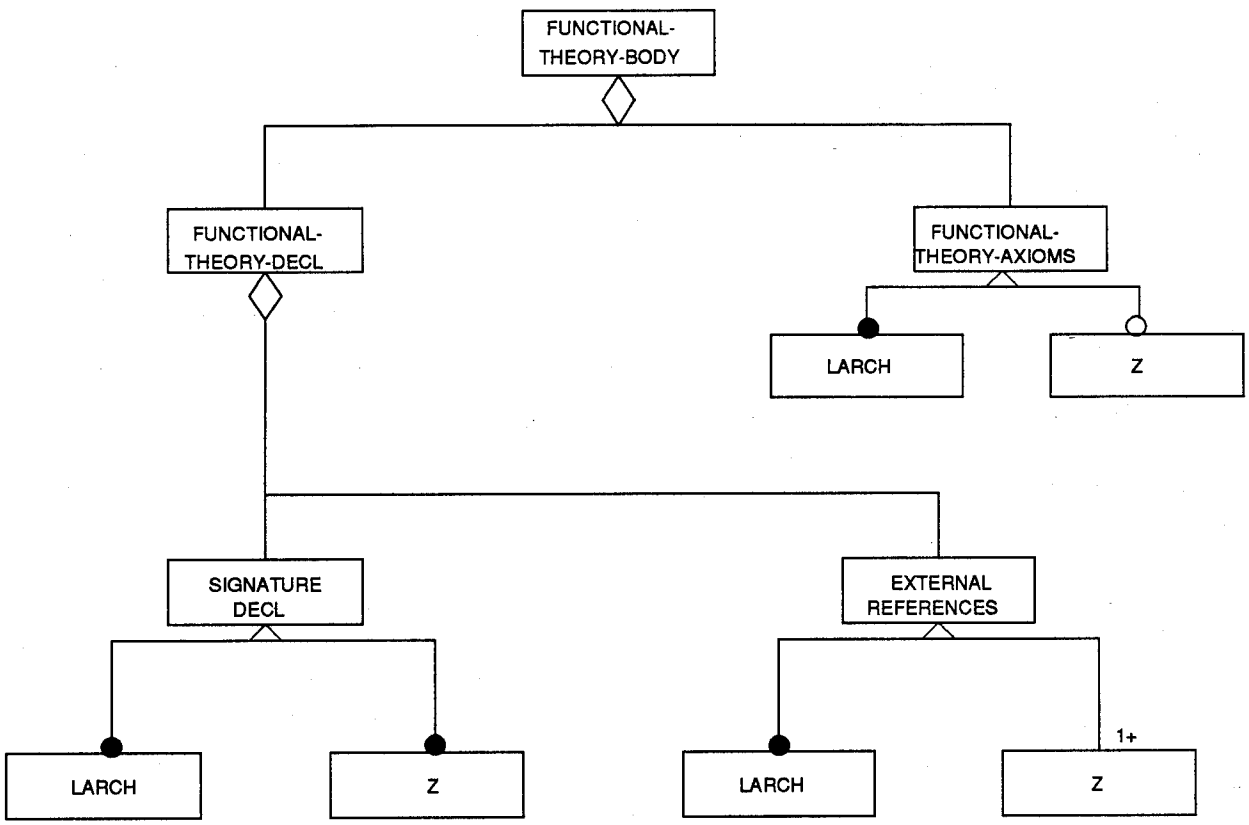


Figure E.4 Unified Functional Theory Body

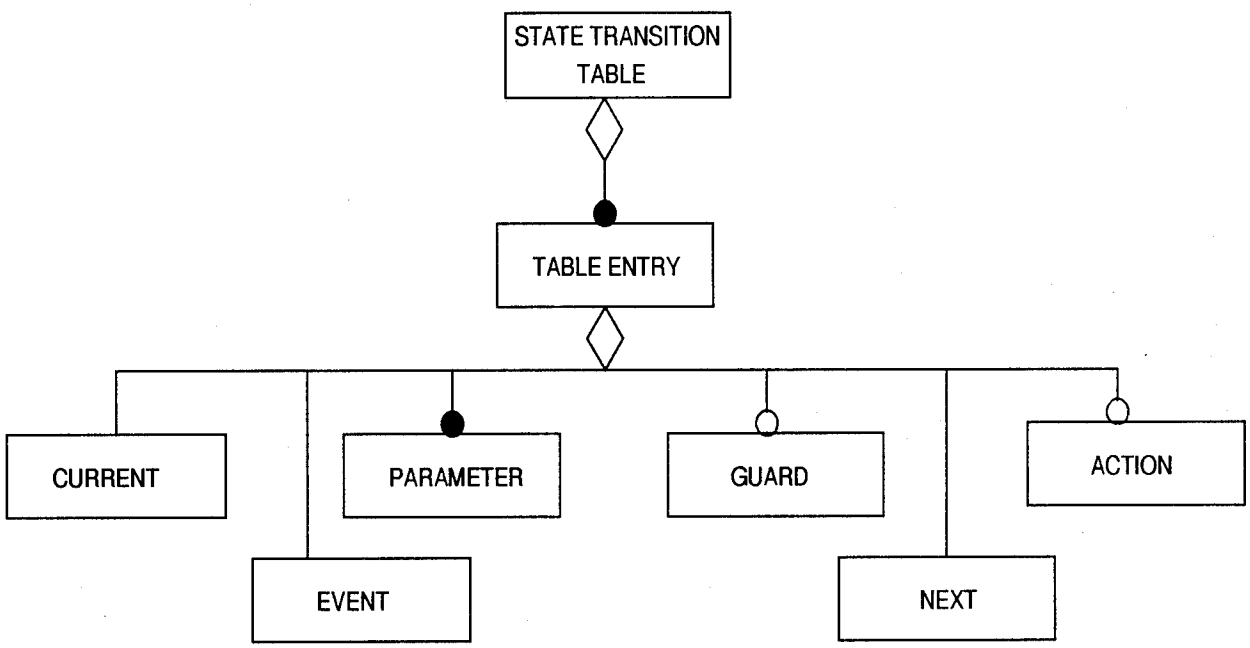


Figure E.5 State Transition Table

Appendix F. REFINE *Code for the Unified Domain Model*

This appendix contains the REFINE code for the unified abstract framework depicted in Appendix E. This code demonstrates the object-based nature of the REFINE specification language. The objects and attributes specified in REFINE are modular and easily extensible using the notion of subclasses and tree-attributes. The flexibility of this specification language makes future enhancements to the unified abstract model straightforward.

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: unified-dm.re

 Description: The following specification defines the Unified Domain
Model objects, attributes, and syntax tree.  The unified domain model
was constructed by analyzing the separate abstract syntax trees (AST)
of Larch and Z.

||#

% ------------------------------------------------------------------------
% Unified Domain Model Object Class Definitions
% ------------------------------------------------------------------------

var Unified-Object          : object-class subtype-of user-object

var DomainTheory            : object-class subtype-of Unified-Object

var DomainTheoryTypes       : object-class subtype-of Unified-Object
  var ObjectTheory          : object-class subtype-of DomainTheoryTypes
  var DynamicTheory         : object-class subtype-of DomainTheoryTypes
  var FunctionalTheory      : object-class subtype-of DomainTheoryTypes

var TheoryId                : object-class subtype-of Unified-Object
  var ObjectTheoryId        : object-class subtype-of TheoryId
  var DynamicTheoryId       : object-class subtype-of TheoryId
  var FunctionalTheoryId    : object-class subtype-of TheoryId

var TheoryBody              : object-class subtype-of Unified-Object
  var ObjectTheoryBody      : object-class subtype-of TheoryBody
  var DynamicTheoryBody     : object-class subtype-of TheoryBody
  var FunctionalTheoryBody  : object-class subtype-of TheoryBody

var TheoryDecl              : object-class subtype-of Unified-Object
  var ObjectTheoryDecl      : object-class subtype-of TheoryDecl
  var DynamicTheoryDecl     : object-class subtype-of TheoryDecl
```

```
  var FunctionalTheoryDecl  : object-class subtype-of TheoryDecl

var SignatureDecl           : object-class subtype-of Unified-Object
var ContextRef              : object-class subtype-of Unified-Object

var TheoryAxioms            : object-class subtype-of Unified-Object
  var ObjectTheoryAxioms    : object-class subtype-of TheoryAxioms
  var DynamicTheoryAxioms   : object-class subtype-of TheoryAxioms
  var FunctionalTheoryAxioms: object-class subtype-of TheoryAxioms

%-------------------------------------------------------------------
% Unified Model Attribute Declarations for Branches in Tree Structure
%-------------------------------------------------------------------
var theory-types            : map(DomainTheory, set(DomainTheoryTypes))
                              computed-using theory-types(x) = {}

var theory-id               : map(DomainTheoryTypes, IdName) = {||}
var ot-id                   : map(DomainTheoryTypes, IdName) = {||}
var dt-id                   : map(DomainTheoryTypes, IdName) = {||}
var ft-id                   : map(DomainTheoryTypes, IdName) = {||}

var theory-body             : map(DomainTheoryTypes, TheoryBody) = {||}
var ot-body                 : map(DomainTheoryTypes, TheoryBody) = {||}
var dt-body                 : map(DomainTheoryTypes, TheoryBody) = {||}
var ft-body                 : map(DomainTheoryTypes, TheoryBody) = {||}

var theory-decl             : map(TheoryBody, TheoryDecl) = {||}
var context-refs             : map(TheoryDecl, set(ContextRef))
                              computed-using context-refs(x) = {}

var signature-decl          : map(TheoryDecl, set(SignatureDecl))
                              computed-using signature-decl(x) = {}

var theory-axioms           : map(TheoryBody, set(TheoryAxioms))
                              computed-using theory-axioms(x) = {}

%-------------------------------------------------------------
% Structure for Abstract Syntax Tree
%-------------------------------------------------------------
form Define-Tree-Attributes-of-Unified-Specification
  Define-Tree-Attributes('DomainTheory, {'theory-types})&
  Define-Tree-Attributes('ObjectTheory, {'ot-id, 'ot-body})&
  Define-Tree-Attributes('DynamicTheory, {'dt-id, 'dt-body})&
  Define-Tree-Attributes('FunctionalTheory, {'ft-id, 'ft-body})&
  Define-Tree-Attributes('TheoryBody, {'theory-decl, 'theory-axioms})&
  Define-Tree-Attributes('TheoryDecl, {'signature-decl, 'context-refs})
%%  Define-Tree-Attributes('TheoryAxioms, {%% fill in Larch/Z specific})&
%%  Define-Tree-Attributes('ContextRef, {%% fill in Larch/Z specific})&
%%  Define-Tree-Attributes('SignatureDecl, {%% fill in Larch/Z specific})&
```

Appendix G. ULARCH *Domain Model*

The LARCH specific extensions to the unified abstract model are shown in the following figures. These models form the basis for future semantic analysis tasks to reduce the language specific notations into a canonical format. This task can be accomplished by extending the unified abstract framework using pattern matching and term rewriting on the LARCH and *Z* specific object classes.

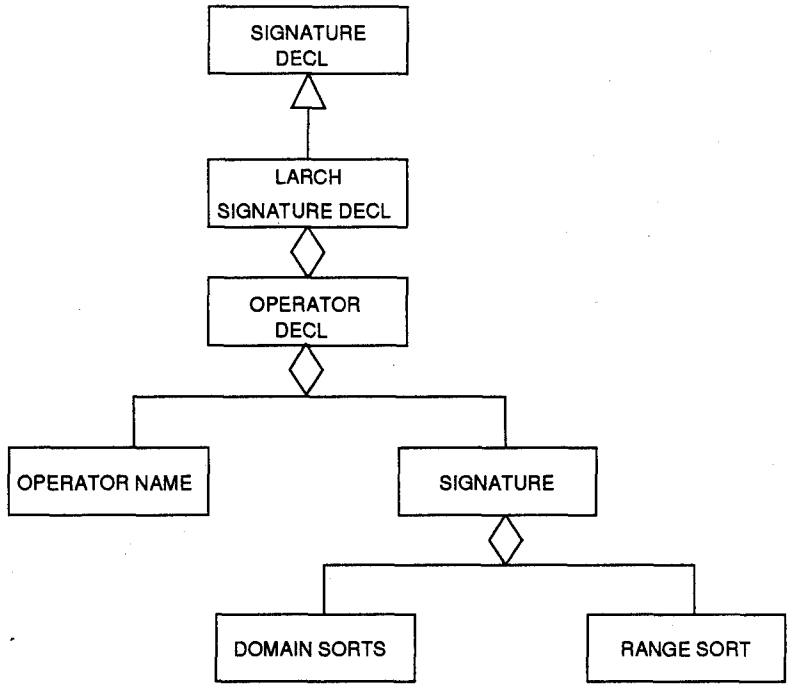


Figure G.1 Domain Model - LARCH Signature Declarations

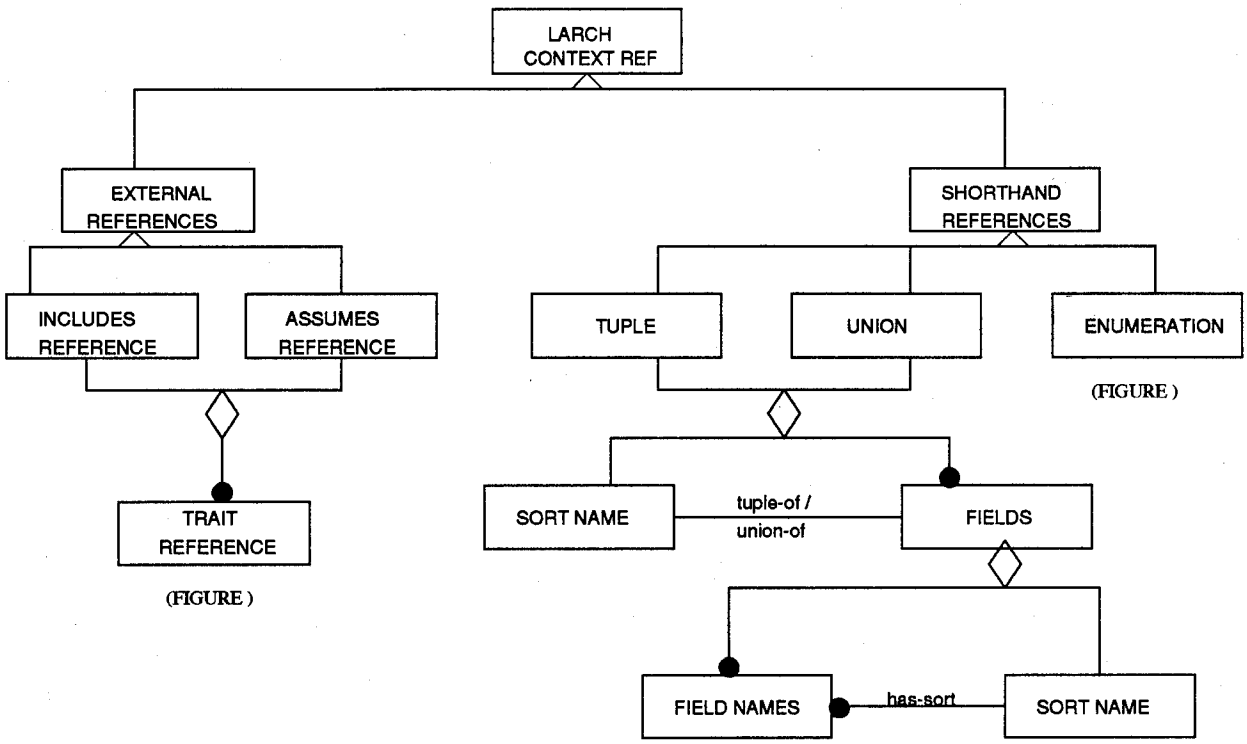


Figure G.2 Domain Model - LARCH Context References

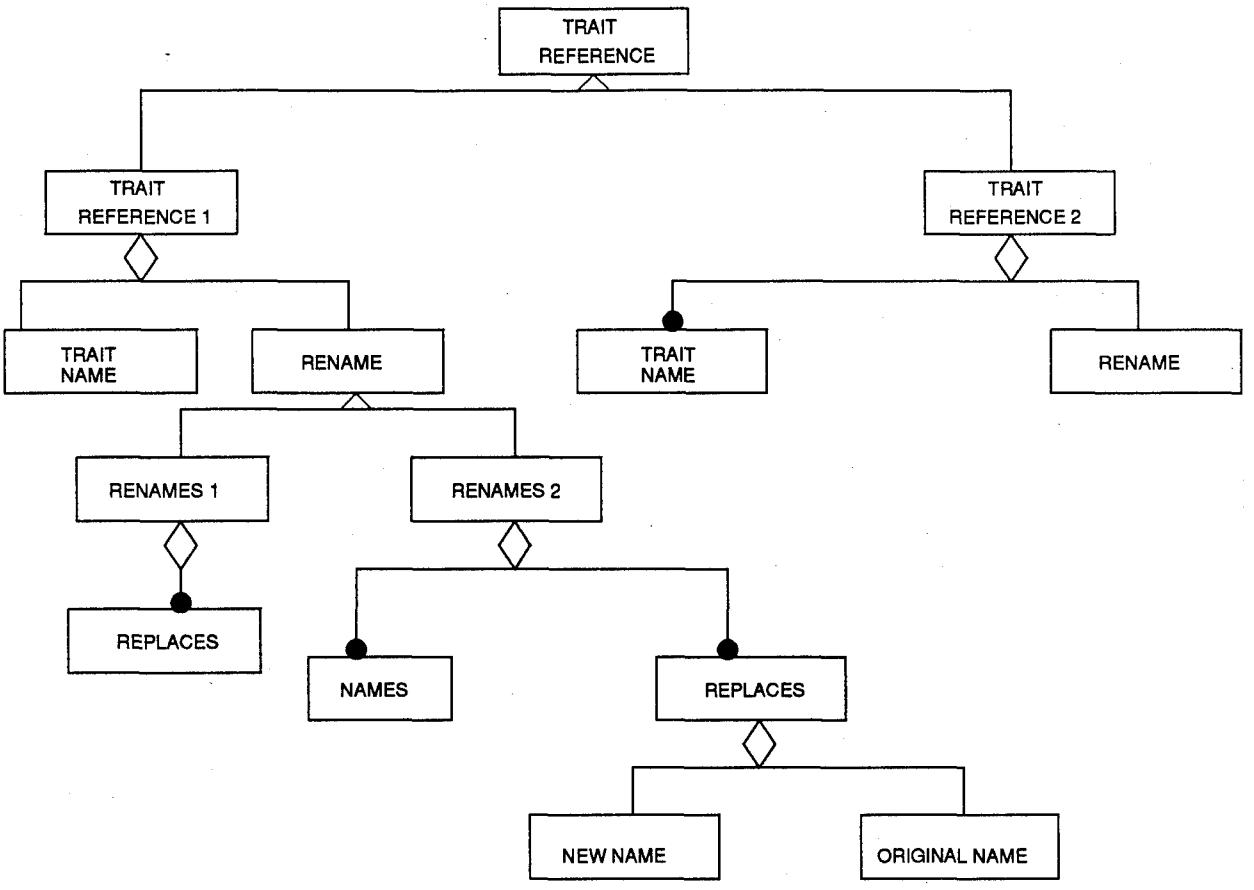


Figure G.3 Domain Model - LARCH Trait References

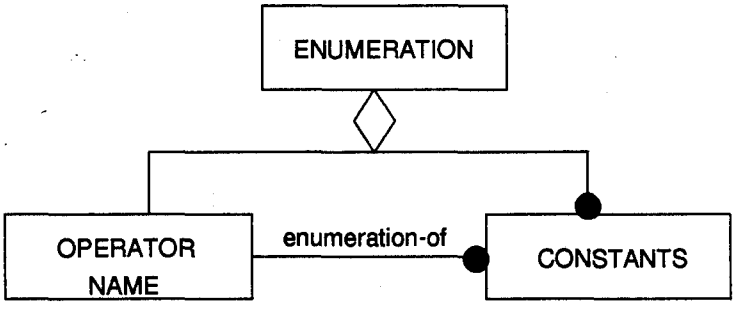


Figure G.4 Domain Model - LARCH Enumeration References

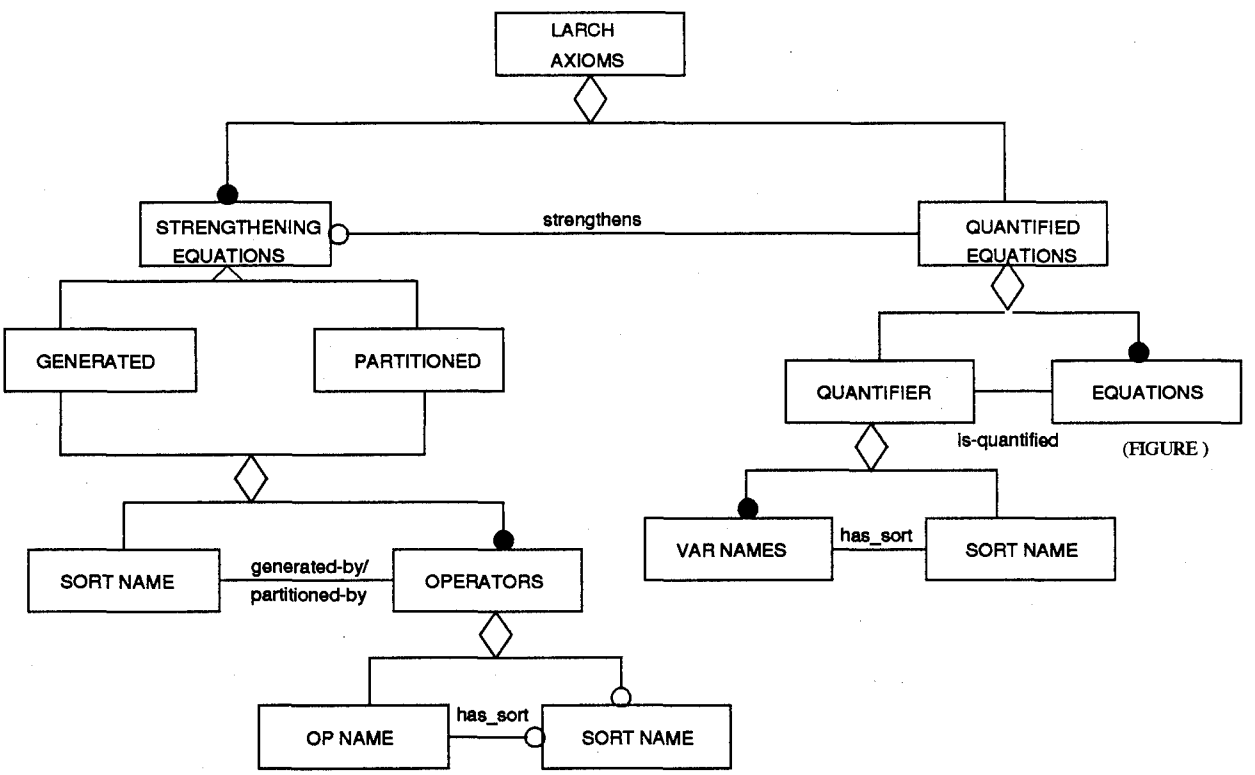


Figure G.5 Domain Model - LARCH Axioms

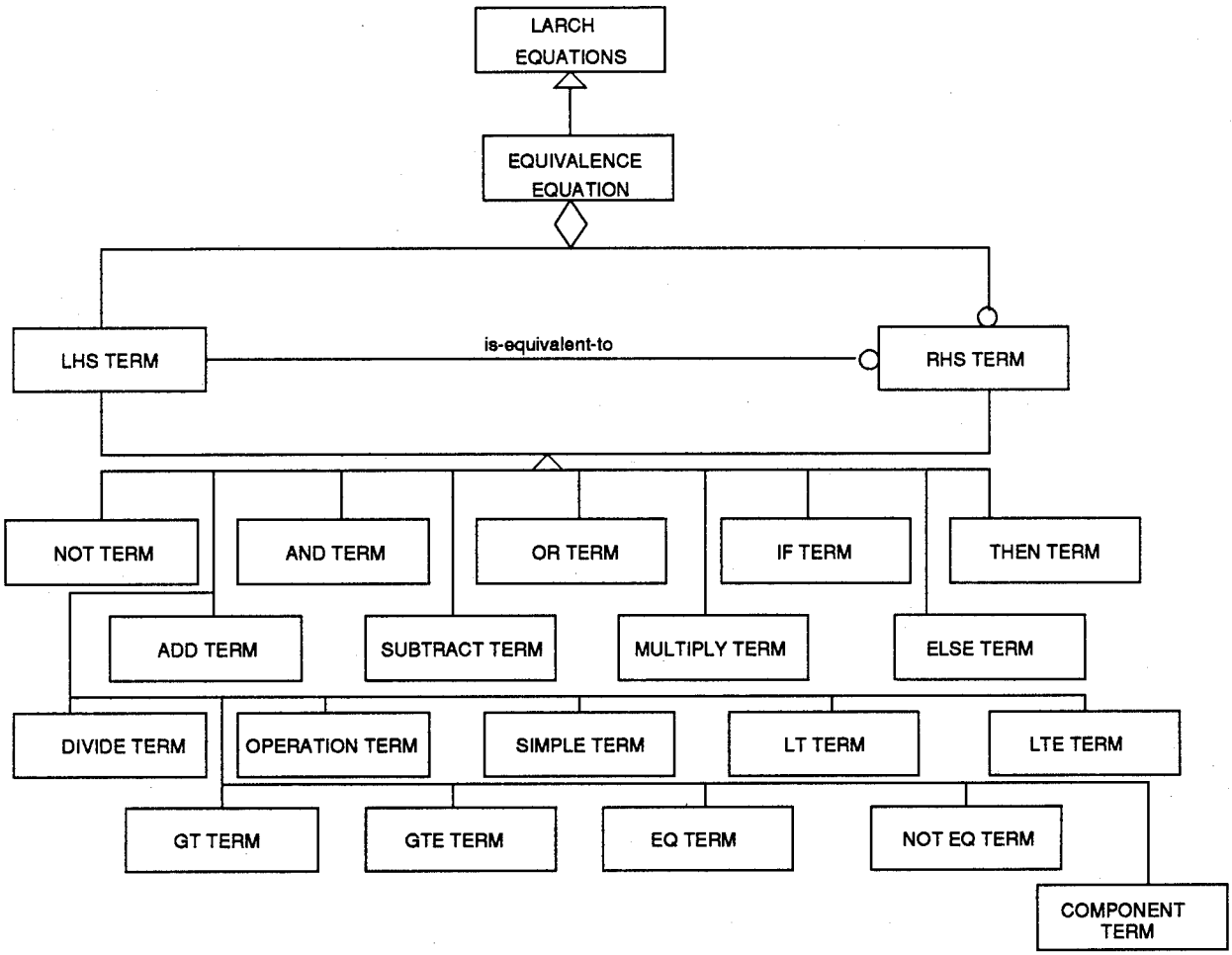


Figure G.6 Domain Model - LARCH Equations

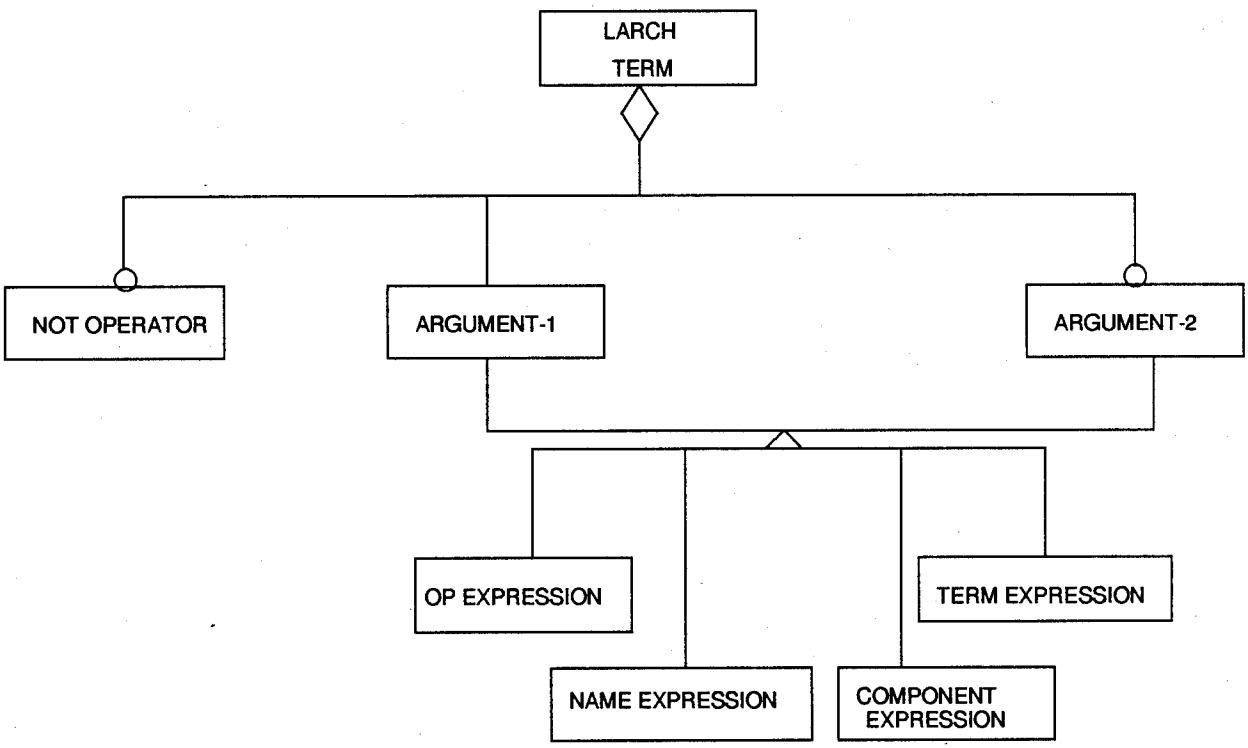


Figure G.7 Domain Model - LARCH Terms

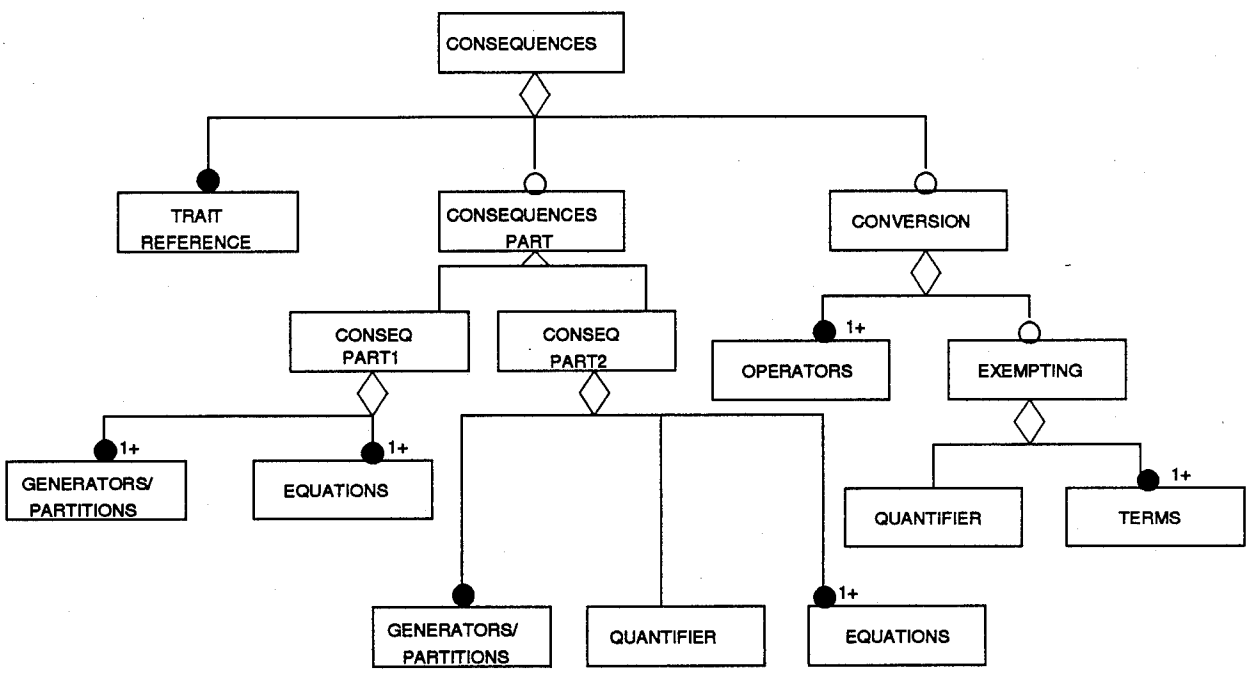


Figure G.8 Domain Model - LARCH Consequences

```
   lteTerm           ::=  [arg-1 "<=" arg-2] builds lteTerm,
   gtTerm            ::=  [arg-1 ">" arg-2] builds gtTerm,
   gteTerm           ::=  [arg-1 ">=" arg-2] builds gteTerm,
   eqTerm            ::=  [arg-1 "=" arg-2] builds eqTerm,
   notEqTerm         ::=  [arg-1 "\\neq" arg-2] builds notEqTerm,

   bracketedExpr     ::=  [open-sym te-rms * "," close-sym {arg-2}] builds bracketedExpr,
   Expr1             ::=  ["(" te-rm ")" {[ "." simple-id]}] builds Expr1,
   Expr2             ::=  [simple-id] builds Expr2,
   Expr3             ::=  [simple-id "(" args + "," ")" {["." simple-id2]}] builds Expr3,
   Expr4             ::=  [sort-id "." field-id] builds Expr4,

   UserOp            ::=  ["\\" op-id] builds UserOp,
   Multiply          ::=  ["*"] builds Multiply,
   Addition          ::=  ["+"] builds Addition,
   Subtraction       ::=  ["-"] builds Subtraction,
   Divide            ::=  ["/"] builds Divide,
   LessThan          ::=  ["<"] builds LessThan,
   LessThanEq        ::=  ["<="] builds LessThanEq,
   Equality          ::=  ["="] builds Equality,
   GreaterThan       ::=  [">"] builds GreaterThan,
   GreaterThanEq     ::= [">="] builds GreaterThanEq,
   And-Obj           ::=  ["\\and"] builds And-Obj,
   Or-Obj            ::=  ["\\or"] builds Or-Obj,
   Not-Obj           ::=  ["~"] builds Not-Obj,
   Implies           ::=  ["\\implies"] builds Implies,
   EqualSym          ::=  ["\\eq"] builds EqualSym,
   NotEqual          ::=  ["\\neq"] builds NotEqual,

%% Consequences Part
   Consequences1     ::=  ["implies" {["(" traitref + "," ")"]}
                           {conseq-part} {con-version}] builds Consequences1,
   ConseqPart1       ::=  [gen-Partitions + "" eq-uations2 + ";"] builds ConseqPart1,

   ConseqPart2       ::=  [gen-Partitions * "" eq-part2] builds ConseqPart2,

   Conversion        ::=  ["converts" op-erators + "," {ex-empting}] builds Conversion,
   Exempting         ::=  ["exempting" {qu-antifier} te-rms + ","] builds Exempting

keyword-alternatives
   "/\\" may-replace "\\and",
   "\\/" may-replace "\\or",
   "=>"  may-replace "\\implies",
   "\\langle" may-replace "\\<",
   "\\rangle" may-replace "\\>",
   "\\geq" may-replace ">=",
   "\\leq" may-replace "<=",
   "\\equals" may-replace "==",
   "\\eqsep" may-replace ";",
   "\\arrow" may-replace "->"
end
```

Appendix J. Semantic Analysis Code

The following Refine program traverses a ULARCH abstract syntax tree (AST) and augments the AST with any included traits by unioning the signatures and the axioms of the included trait with the trait doing the inclusion. The end result is an expanded AST that defines all of a specification's appropriate signatures and axioms. This expanded AST simplifies the semantic analysis and code generation tasks.

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: includes.re

||#

!! in-package("RU")
!! in-grammar('user)

function make-includes-link ()  =
  let (Temp-Op-Decl  : OpDecl = make-object('OpDecl))
  let (specs         : set(DomainTheoryTypes) =
                       instances(find-object-class('DomainTheoryTypes), true))
  let (Temp-Symbol   : Symbol = newsymbol('TempSymbol))
  let (Temp-Theory-Axiom  : TheoryAxioms = make-object('TheoryAxioms))
  let (Temp-Tuple-Obj     : Tuple-Obj = make-object('Tuple-Obj))

   (enumerate tr over specs do
    (enumerate i over descendants-of-class(theory-body(tr), 'Includes) do
     (enumerate j over descendants-of-class(i, 'IdName) do
      (Temp-Symbol <- name(j);
       enumerate k over specs do
       (if (Temp-Symbol = name(theory-id(k))) then
(enumerate l over descendants-of-class(theory-body(k), 'OpDecl) do
 (Temp-Op-Decl <- l;
  op-decls(signature-decl(theory-decl(theory-body(tr)))) <-
  op-decls(signature-decl(theory-decl(theory-body(tr)))) with Temp-Op-Decl
  )
 );
(enumerate m over descendants-of-class(theory-body(k), 'TheoryAxioms) do
 (Temp-Theory-Axiom <- m;
  theory-axioms(theory-body(tr)) <- theory-axioms(theory-body(tr)) with
  Temp-Theory-Axiom
  )
 );
(enumerate n over descendants-of-class(theory-body(k), 'Tuple-Obj) do
 (Temp-Tuple-Obj <- n;
```

```
  context-refs(theory-decl(theory-body(tr))) <-
  context-refs(theory-decl(theory-body(tr))) with Temp-Tuple-Obj
  )
 )
)
       )
      )
     )
    )
```

Appendix K. Target Domain Model

This appendix contains the object-oriented domain model for the execution framework. REFINE is the target framework, and the following figures identify the principal object classes. This model guides the development of the execution framework mappings providing a concrete target for the source objects.

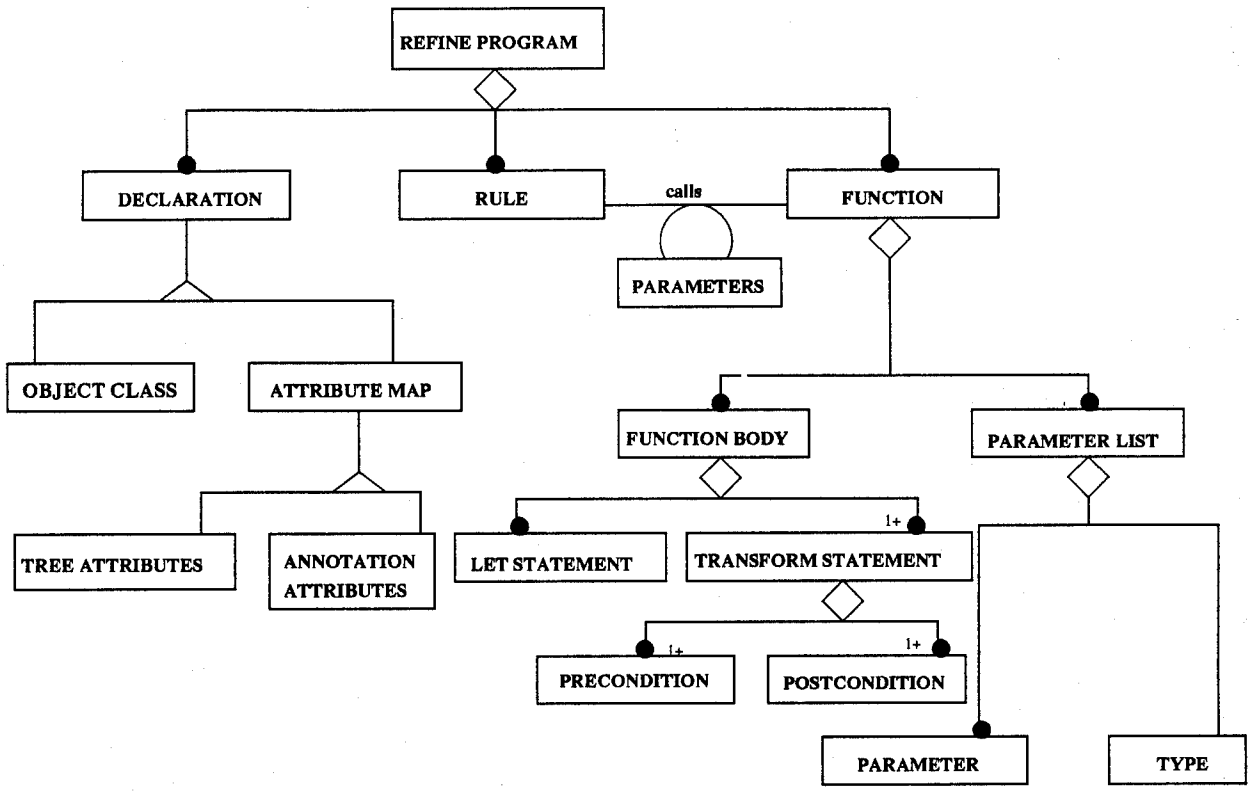


Figure K.1 Unified Domain Theory Model - 1

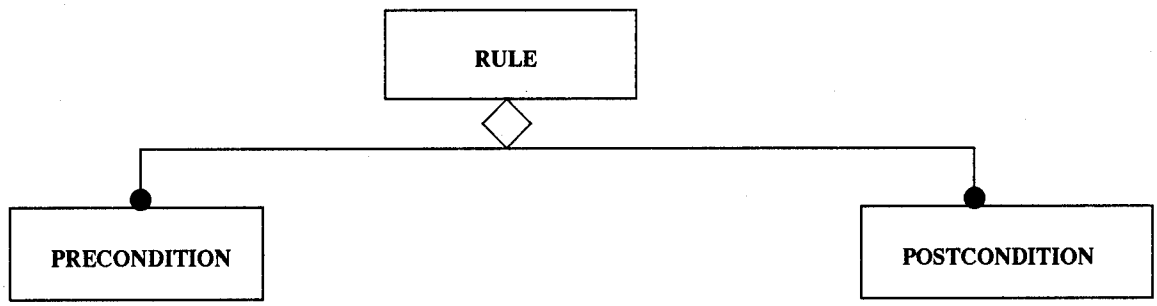


Figure K.2 Unified Domain Theory Model - 2

Appendix L. REFINE *Code for the Target Domain Model*

The following REFINE specification defines the object classes and map attributes for the execution framework. Because the target execution framework is REFINE, the code incorporates the notions of object classes, map attributes, functions, and rules, as the target objects.

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: exec.re

||#

%-------------------------------------------------------------
% Refine Object Class Declaration
%-------------------------------------------------------------

var Refine-Object-Model              : object-class subtype-of user-object
var Refine-Declaration               : object-class subtype-of Refine-Object-Model

var Refine-Decl-Attrs                : object-class subtype-of Refine-Object-Model
  var Refine-Annotation-Attr         : object-class subtype-of Refine-Decl-Attrs
  var Refine-Tree-Attr               : object-class subtype-of Refine-Decl-Attrs
var Refine-Decl-Obj-Class            : object-class subtype-of Refine-Object-Model

var Refine-Function                  : object-class subtype-of Refine-Object-Model
var Refine-Function-Body             : object-class subtype-of Refine-Object-Model
var Function-Parameters              : object-class subtype-of Refine-Object-Model
var Parameter-Value                  : object-class subtype-of Refine-Object-Model
var Parameter-Type                   : object-class subtype-of Refine-Object-Model
var Function-Let-Statement           : object-class subtype-of Refine-Object-Model
var Function-Transform               : object-class subtype-of Refine-Object-Model
var Function-Precondition            : object-class subtype-of Refine-Object-Model
var Function-Postcondition           : object-class subtype-of Refine-Object-Model

var Refine-Rule                      : object-class subtype-of Refine-Object-Model
var Rule-Precondition                : object-class subtype-of Refine-Object-Model
var Rule-Postcondition               : object-class subtype-of Refine-Object-Model

%-------------------------------------------------------------
% Refine Attribute Declarations for Branches in Tree Structure
%-------------------------------------------------------------
var refineDecl                       : map(Refine-Object-Model, Refine-Declaration) = {||}
var refineObjClasses                 : map(Refine-Object-Model, set(Refine-Decl-Obj-Class))
                                       computed-using refineObjClasses(x) = {}
```

```
var refineAttrs                          : map(Refine-Object-Model, set(Refine-Decl-Attrs))
                                           computed-using refineAttrs(x) = {}

var refineFunctions                      : map(Refine-Object-Model, set(Refine-Function))
                                           computed-using refineFunctions(x) = {}
var functionBody                         : map(Refine-Object-Model, Refine-Function-Body) = {||}
var parameterList                        : map(Refine-Object-Model, set(Function-Parameters))
                                           computed-using parameterList(x) = {}
var parameters                           : map(Function-Parameters, set(Parameter-Value))
                                           computed-using parameters(x) = {}
var parameterType                        : map(Function-Parameters, Parameter-Type) = {||}
var letStatements                        : map(Refine-Object-Model, set(Function-Let-Statement))
                                           computed-using letStatements(x) = {}
var transforms                           : map(Refine-Object-Model, set(Function-Transform))
                                           computed-using transforms(x) = {}
var functionPre                          : map(Refine-Object-Model, set(Function-Precondition))
                                           computed-using functionPre(x) = {}
var functionPost                         : map(Refine-Object-Model, set(Function-Postcondition))
                                           computed-using functionPost(x) = {}

var refineRules                          : map(Refine-Object-Model, set(Refine-Rule))
               computed-using refineRules(x) = {}
var rulePre                              : map(Refine-Object-Model, set(Rule-Precondition))
               computed-using rulePre(x) = {}
var rulePost                             : map(Refine-Object-Model, set(Rule-Postcondition))
                                           computed-using rulePost(x) = {}

%-------------------------------------------------------------
%  Annotation Attributes
%-------------------------------------------------------------
%% For Object Refine-Decl-Attrs
var attrDomain                            : map(Refine-Decl-Attrs, symbol) = {||}
var attrCoDomain                          : map(Refine-Decl-Attrs, symbol) = {||}

%-------------------------------------------------------------
%  Declarations for other Objects
%-------------------------------------------------------------

var Exec-Program          : Refine-Object-Model = make-object('Refine-Object-Model)

%-------------------------------------------------------------
%  Structure for Abstract Syntax Tree
%-------------------------------------------------------------

form Define-Tree-Attributes-of-Refine-Rules
  Define-Tree-Attributes('Refine-Object-Model, {'refineDecl, 'refineFunctions,
'refineRules})&
  Define-Tree-Attributes('Refine-Declaration, {'refineObjClasses,
       'refineAttrs})&
  Define-Tree-Attributes('Refine-Function, {'parameterList, 'functionBody})&
  Define-Tree-Attributes('Function-Parameters, {'parameters, 'parameterType})&
```

```
Define-Tree-Attributes('Refine-Function-Body, {'letStatements,
'transforms})&
Define-Tree-Attributes('Function-Transform, {'functionPre, 'functionPost})&
Define-Tree-Attributes('Refine-Rule, {'rulePre, 'rulePost})
```

Appendix M. REFINE *Code for the Initial Execution Code*

The following REFINE program creates an initial framework for an executable program. The main function, Make-Executable, encapsulates the Make-Object-Classes, Make-Attributes, and Make-Function operations generating the appropriate object classes and attributes in the execution framework from the source objects. After creating the execution framework, the Print-Program function traverses the executable AST and prints an executable program's initial shell consisting of the object classes, map attributes, function names, and parameters.

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: transform.re

||#

function Make-Object-Class (Temp-Specs        : set(DomainTheoryTypes),
    Temp-Decl         : Refine-Declaration,
    Temp-Exec-Program : Refine-Object-Model) : Refine-Object-Model =

  let (Refine-Object      : Refine-Decl-Obj-Class = make-object('Refine-Decl-Obj-Class))

  (enumerate tr over Temp-Specs do
   (enumerate o over descendants-of-class(tr, 'ObjectTheory) do
    Refine-Object <- set-attrs(make-object('Refine-Decl-Obj-Class),
       'Name, name(theory-id(o)));
    (format (true, "Refine-Object ~a~%", Refine-Object));
    refineObjClasses(Temp-Decl) <- refineObjClasses(Temp-Decl) with
                                        Refine-Object;
    (format (true, "programDecl ~a~%", Temp-Decl))
    )
   );
   refineDecl(Temp-Exec-Program) <- Temp-Decl


function Make-Attributes (Temp-Specs        : set(DomainTheoryTypes),
  Temp-Decl         : Refine-Declaration,
  Temp-Exec-Program : Refine-Object-Model) : Refine-Object-Model =

  let (Refine-Attr       : Refine-Decl-Attrs = make-object('Refine-Decl-Attrs))

  (enumerate tr over Temp-Specs do
   (enumerate o over descendants-of-class(tr, 'ObjectTheory) do
```

```
    (enumerate a over descendants-of-class(o, 'Tuple-Obj) do
     (enumerate f over descendants-of-class(a, 'Field) do
      (enumerate fi over field-ids(f) do
       Refine-Attr <- set-attrs(make-object('Refine-Decl-Attrs),
'Name, name(fi),
'attrDomain, name(theory-id(tr)),
'attrCoDomain, name(sort-id(f)));

       (format (true, "Refine-Attr ~a~%", Refine-Attr));
       refineAttrs(Temp-Decl) <- refineAttrs(Temp-Decl) with
       Refine-Attr;
       (format (true, "programDecl ~a~%", Temp-Decl))
       )
      )
     )
    )
   );
   refineDecl(Temp-Exec-Program) <- Temp-Decl;
   (format (true, "Exec-Program ~a~%", Temp-Exec-Program))

function Make-Function (Temp-Specs        : set(DomainTheoryTypes),
Temp-Exec-Program : Refine-Object-Model) : Refine-Object-Model =

  let (Temp-Function   : Refine-Function = make-object('Refine-Function))
  let (Temp-Set        : set(Refine-Function) = {})
  let (Temp-Param      : Parameter-Value = make-object('Parameter-Value))
  let (Temp-Param-Set  : set(Parameter-Value) = {})
  let (Temp-Param-List : set(Function-Parameters) = {})

  (enumerate tr over Temp-Specs do
   (enumerate d over descendants-of-class(tr, 'DomainTheoryTypes) do
    (enumerate e over descendants-of-class(d, 'OpDecl) do
     (enumerate f over descendants-of-class(e, 'NameObj) do
       if defined?(op-id(f)) then
         Temp-Function <- set-attrs(make-object('Refine-Function),
    'Name, name(op-id(f)));
         format(true, "Temp-Function ~a~%", Temp-Function);
         if empty(Temp-Set) then
           (Temp-Set <- Temp-Set with Temp-Function
    )
         else
           (enumerate g over Temp-Set do
    if name(g) ~= name(Temp-Function) then
              Temp-Function
    else
      Temp-Function <- undefined
    );
        if defined?(Temp-Function) then
          Temp-Set <- Temp-Set with Temp-Function
      );
     (enumerate s over descendants-of-class(e, 'SortSet) do
```

```
      (format(true, "in SortSet ~%");
       enumerate d over domain-ids(s) do
       Temp-Param <- d;
       format(true, "temp-param is ~a~%", Temp-Param);
       if empty(Temp-Param-Set) then
         Temp-Param-Set <- Temp-Param-Set with Temp-Param
       else
         (enumerate x over Temp-Param-Set do
  if name(x) ~= name(Temp-Param) then
    Temp-Param
  else
    Temp-Param <- undefined
  );
       if defined?(Temp-Param) then
         Temp-Param-Set <- Temp-Param-Set with Temp-Param
       )
      )
     )
    )
   );
   refineFunctions(Temp-Exec-Program) <- refineFunctions(Temp-Exec-Program) union Temp-Set;
   (format (true, "Exec-Program ~a~%", Temp-Exec-Program))


function Make-Executable (theory : DomainTheory) : Refine-Object-Model =

  let (Specs               : set(DomainTheoryTypes) = {})
  let (Temp-Exec-Program   : Refine-Object-Model = make-object('Refine-Object-Model))
  let (Program-Decl        : Refine-Declaration = make-object('Refine-Declaration))

  (enumerate s over descendants-of-class(theory, 'DomainTheoryTypes) do
   Specs <- Specs with s
   );
  (Make-Object-Class(Specs, Program-Decl, Temp-Exec-Program));
  (Make-Attributes(Specs, Program-Decl, Temp-Exec-Program));
  (Make-Function(Specs, Temp-Exec-Program))

function Print-Program (Program : object) =

  undefined?(Program) --> format(true, "Nothing to print ~%");
  defined?(Program) -->
    format (true, "!! in-package(RU) ~%");
    format (true, "!! in-grammar('user) ~% ~% ~%");
    (enumerate d over descendants-of-class(Program, 'Refine-Declaration) do
     (enumerate o over descendants-of-class(d, 'Refine-Decl-Obj-Class) do
        format(true, "%---------------- ~A ----------------------% ~%", name(o));
        format(true, "%-------------------------------------------------% ~%");
        format(true, "var ~A : object-class subtype-of user-object ~%",
       name(o));
        format(true, " ~%")
      );
```

```
  (enumerate a over descendants-of-class(d, 'Refine-Decl-Attrs) do
     format(true, "var ~A    : map(~A, ~A) = {||} ~%", name(a), attrDomain(a),
    attrCoDomain(a));
     format(true, " ~%")
   )
  );
(enumerate e over descendants-of-class(Program, 'Refine-Function) do
 format(true, "function ~A () = ~%", name(e))
 )
```

Appendix N. User's Manual for the Unified LARCH *Parser*

This appendix contains a listing of the REFINE files required to parse LARCH traits. The order in which the files are listed below indicates the required compilation order. Following the file listings is a set of instructions guiding a user through a sample session for parsing LARCH traits and viewing the corresponding AST. In addition, the final instruction shows the user how to create an execution framework for the parsed LARCH traits.

```
1. Load system files for Dialect and Object Inspector.

  (load ''load-inspector'')
  (load-inspector)

2. Load the unified {\sc Larch} domain model and grammar files and
   the target execution domain model.  Bring the unified {\sc Larch}
   grammar in the environment.

  (load ''ularch-dm'')
  (load ''ularch-gram'')
  (in-grammar 'ularch)
  (load ''exec'')

3. Load the semantic analysis file.

  (load ''includes.fasl4'')

4. Load the translation program.
  (load ''transform.fasl4'')

5. Parse your domain specification.
  % C-x-i ~/research/demo/fueltank_domain/ut/combined.ut (for fuel tank domain)
  % C-x-i ~/research/demo/counter_domain/refine/ucounter.re (for counter domain)

6. View the initial domain specification using {\sc Object Inspector}.

  (pup)                                          % returns the magic object number

  (insp::inspect(mcn <object's magic-number>)) % brings {\sc Object Inspector}
                                                 % up with the current object.
  %% for demo view a functional trait from the Counter and Fuel Tank Domains
```

7. Execute ''make-includes-link'' on your domain specification
   to bring in all referenced specifications.

   (make-includes-link(mcn <object's magic-number>))

8. View the augmented AST using {\sc Object Inspector}.

   (insp::inspect-obj(mcn <object's magic-number>))
   %% for demo view a functional trait from the Counter and Fuel Tank Domains

9. Execute the translation function.

   (make-executable(mcn <domaintheory object's magic number>))
   (print-program (mcn <executable object's magic number>))

10. Example of Demonstration Sequence

    a. load the counter domain % C-x-i ~/research/demo/counter_domain/refine/ucounter.re
    b. parse the domain theory
    c. show the domain's AST % (pup) to get domain theory's object number
                             % (insp::inspect(mcn <object's magic-number>))
    d. perform short-hand expansion % (make-includes-link)
    e. perform execution translation % (make-executable)
    f. print executable shell % (print-program (mcn <executable object's
                                                magic number>))
    g. other test domains: fuel tank and SimpleGraph.re
            % file location (~/research/demo/fueltank_domain/ut/combined.ut)
            % file location (~/research/demo/lsl/test/SimpleGraph.re)

Appendix O. Validation Specifications

This appendix contains the sample set of MIT traits that have been parsed successfully. In addition, Major Scott Deloach's graph theory has been parsed and is included here. Parsing these traits demonstrates that the parser can handle a wide-range of LARCH specifications. Note the placement of parentheses for compound terms in the algebraic equations. These parentheses are necessary to reduce the ambiguity during parsing.

O.1 MIT Sample Traits

```
 %ObjectTheory
ListStructure(A, E, C): trait
includes List
E union of list: C, atom: A
```

```
 %ObjectTheory
PQueue (E, Q): trait
  assumes TotalOrder (E)
  includes Container(Q for C)
  asserts ∀ e, e1: E, q: Q
    head(insert(e, q)) ==
      if isEmpty(q) ∨ (e > head(q))
        then e else head(q);
    tail(insert(e, q)) ==
      if isEmpty(q) ∨ (e > head(q))
        then q else insert(e, tail(q))
  implies
    ∀ q: Q, e: E
      (e ∈ q) ⇒ ¬(e < head(q))
    converts head, tail, isEmpty, count, ∈
      exempting head(empty), tail(empty)
```

```
 %ObjectTheory
PairwiseSum(C): trait
  assumes Container(Int, C)
  includes Integer, PairwiseExtension(+, ⊕, Int, C)
  implies (Associative(⊕, C),
          Commutative(⊕ for ∘, C for T, C for Range))
```

O.2 Graph Theory[1]

```
 %ObjectTheory
SimpleGraph (E, V, G): trait
```

[1]This theory was written by Major Scott Deloach.

```
G tuple of v : VSet, e : ESet
includes Set(V for E, VSet for C)
includes Set(E for E, ESet for C)
E tuple of v1: V, v2 : V
includes GraphRelation (E, V, G, R)
assumes Symmetric(R)

introduces
  vertices: G → VSet
  edges: G → ESet
  complete, forest, existsCircuit, noLoops, existsLoops,
  tree, connected: G → Bool
  subgraph: G, G → Bool
  endVertex: V, G → Bool
  adjacent, existsPath: V, V, G → Bool
  insertVertex: V, G → G
  insertEdge: V, V, G → G
  numEdges, numVertices: G → Int
  degree: V, G → Int

asserts ∀ v, v1, v2: V, g, g1: G
  vertices(g) == g.v;
  edges(g) == g.e;
  endVertex(v, g) == ((([v, v1]) ∈ g.e) ∧ (([v, v2]) ∈ g.e)) ⇒ (v1 = v2);
  adjacent(v1, v2, g) == ([v1, v2]) ∈ g.e;
  existsPath(v1, v2, g) == v1 ⟨ (R(g)*) ⟩ v2;
  connected(g) == ((v1 ∈ g.v) ∧ (v2 ∈ g.v)) ⇒ (existsPath(v1, v2, g));
  complete(g) == ((v1 ∈ g.v) ∧ (v2 ∈ g.v)) ⇒ (([v1, v2]) ∈ g.e);
  forest(g) == (v1 ∈ g.v) ⇒ (¬ (v1 ⟨ (R(g)+) ⟩ v1));
  existsCircuit(g) == ¬ forest(g);
  noLoops(g) == (v ∈ g.v) ⇒ (([v, v]) ∉ g.e);
  existsLoops(g) == ¬ noLoops(g);
  tree(g) == existsPath(v1, v2, g) ∧ forest(g);
  subgraph(g1, g) == (([v1, v2]) ∈ g1.e) ⇒ (([v1, v2]) ∈ g.e);
  insertVertex(v, g) == [(g.v ∪ ({v})), g.e];
  insertEdge(v1, v2, g) == [g.v, (g.e ∪ (({[v1, v2]}) ∪ ({[v2, v1]})))];

  numEdges(g) == div(size(g.e), 2);
  numVertices(g) == size(g.v);
  degree(v, g) == size(rangeRestrict(R(g), [v, v]))

implies
  ∀ v, v1, v2: V, g1, g: G
    vertices([{}, {}]) == {};
    adjacent(v1, v2, ([{}, {}])) == false;
    existsPath(v1, v2, ([{}, {}])) == false;
    connected([{}, {}]) == true;
    complete([{}, {}]) == true;
    forest([{}, {}]) == true;
    existsLoops([{}, {}]) == false;
    existsCircuit([{}, {}]) == false;
    noLoops([{}, {}]) == true;
    tree([{}, {}]) == true;
```

```
subgraph([{}, {}], g) == true;
insertEdge(v1, v2, [{}, {}]) == [(({v1}) ∪ ({v2})), (({[v1, v2]}) ∪ ({[v2, v1]}))];
insertVertex(v, ([{}, {}])) == [({v}), {}];
numEdges([{}, {}]) == 0;
numVertices([{}, {}]) == 0;

vertices(insertEdge(v1, v2, g)) == g.v ∪ (({v1}) ∪ ({v2}));
adjacent(v1, v2, insertEdge(v1, v2, g)) == true;
existsPath(v1, v2, insertEdge(v1, v2, g)) == true;
connected(insertEdge(v1, v2, g)) ==
        connected(g) ∧ ((v1 ∈ g.v) ∨ (v2 ∈ g.v));
complete(insertEdge(v1, v2, g)) == (v1 ∈ g.v) ∧ (v2 ∈ g.v);
forest(insertEdge(v1, v2, g)) ⇒ (¬ existsCircuit(g));
noLoops(insertEdge(v1, v2, g)) == noLoops(g) ∧ (v1 ≠ v2);
existsLoops(insertEdge(v1, v2, g))  == existsLoops(g) ∨ (v1 = v2);
tree(insertEdge(v1, v2, g)) ⇒ (¬ (existsCircuit(g) ∧ (¬ tree(g))));
subgraph(g1, insertEdge(v1, v2, g)) == subgraph(g1, g);
subgraph(insertEdge(v1, v2, g1), g) == subgraph(g1, g) ∧ (([v1, v2]) ∈ g.e);
insertEdge(v1, v2, insertEdge(v1, v2, g)) == insertEdge(v1, v2, g);
numEdges(insertEdge(v1, v2, g)) == if (([v1, v2]) ∈ g.e)
                                then numEdges(g) else numEdges(g) + 1;
numVertices(insertEdge(v1, v2, g)) == numVertices(g) +
                                  (((if v1 ∈ g.v then 0 else 1)) +
                                  ((if v2 ∈ g.v then 0 else 1)));
degree(v, insertEdge(v1, v2, g)) == degree(v, g) +
                                 (if ((v = v1) ∧ (v = v2)) then 2
                                  else (if ((v = v1) ∨ (v = v2)) then 1
                                  else 0));

vertices(insertVertex(v, g)) == g.v ∪ ({v});
adjacent(v1, v2, insertVertex(v, g)) == adjacent(v1, v2, g);
connected(insertVertex(v, g)) == connected(g) ∧ (v ∈ g.v);
complete(insertVertex(v, g)) == complete(g) ∧ (v ∈ g.v);
forest(insertVertex(v, g)) == forest(g);
noLoops(insertVertex(v, g)) == noLoops(g);
existsLoops(insertVertex(v, g))  == existsLoops(g);
tree(insertVertex(v, g)) == tree(g) ∧ (v ∈ g.v);
subgraph(g1, insertVertex(v, g)) == subgraph(g1, g)
                                    ∨ subgraph(([(g1.v - ({v})), g.e]), g);
insertEdge(v1, v2, insertVertex(v, g)) == insertVertex(v, insertEdge(v1, v2, g));
numEdges(insertVertex(v, g)) == numEdges(g);
numVertices(insertVertex(v, g)) == if v ∈ g.v then numVertices(g)
                                                 else numVertices(g) + 1;
degree(v1, insertVertex(v, g)) == degree(v1, g);

(connected(g) ∧ ((v1 ∈ g.v) ∧ (v2 ∈ g.v)))
             ⇒ existsPath(v1, v2, g);
(complete(g) ∧ ((v1 ∈ g.v) ∧ (v2 ∈ g.v)))
             ⇒ adjacent(v1, v2, g);
(tree(g) ∧ ((v1 ∈ g.v) ∧ (v2 ∈ g.v)))
             ⇒ existsPath(v1, v2, g);
(tree(g) ∧ (((v1 ∈ g.v) ∧ (v2 ∈ g.v))
              ∧ (¬adjacent(v1, v2, g))))
```

```
                    ⇒ existsCircuit(insertEdge(v1, v2, g));
tree(g) ⇒ (¬ existsCircuit(g));
tree(g) ⇒ (numEdges(g) = (numVertices(g) - 1));
tree(g) ⇒ connected(g);
((degree(v, g) ≥ 2) ∧ (v ∈ g.v)) ⇒ existsCircuit(g);
complete(g) ⇒ (numEdges(g) = ((numVertices(g) * (numVertices(g) - 1)) / 2));
endVertex(v, g) == (degree(v, g) = 1);
existsLoops(g) ⇒ existsCircuit(g)
```

And92. Cindy Anderson. Creating and Manipulating Formalized Software Architectures to Support a Domain-Oriented Application Composition System. Master's thesis, Air Force Institute of Technology, 1992.

ASU88. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1988.

Bai. Paul D. Bailor. Theories and Software Engineering. Class Notes: CSCE 793, Winter 1994, Air Force Institute of Technology.

BFG+94. Lee Blaine, Rafael Furst, Li-Mei Gilham, Allen Goldberg, Richard Jullig, Jim McDonald, and Y.V. Srinivas. *Specware User Manual.* Reasoning Systems, October 1994.

Bre91. Ruth Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments.* Number 562 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

EMCO92. Ehrig, Mahr, Classen, and Orejas. Introduction to Algebraic Specification. Part 1: Formal Methods for Software Development. *The Computer Journal*, 35(5):460–467, 1992.

EMO92. Ehrig, Mahr, and Orejas. Introduction to Algebraic Specification. Part 2: From Classical View to Foundations of System Specifications. *The Computer Journal*, 35(5):468–476, 1992.

FGJM85. Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and Jose Meseguer. *Principles of OBJ2.* Ventura Hall, February 1985.

Ger93. Captain Mark Gerken. *Dissertation Prospectus: Specification and Design Theories for Software Architectures.* Dissertation prospectus, Air Force Institute of Technology, September 1993.

GH93. John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

GT79. Joseph A. Goguen and Joseph J. Tardo. An Introduction to OBJ. In *IEEE 1979 Specifications of Reliable Software*, pages 170–189, 1979.

Hal90. Anthony Hall. Seven Methods of Formal Methods. *IEEE Software*, pages 11–19, 1990.

HB94. Thomas C. Hartrum and Paul D. Bailor. Teaching Formal Extensions of Informal-Based Object-Oriented Analysis Methodologies. In *Software Engineering Education Proceedings*, Pittsburgh, PA, January 1994. Software Engineering Education, SEI, Software Engineering Institute (SEI).

Hun94. Robert Hunt. Modeling Operational Task Assignment in Air Force Wing Command and Control System. Master's thesis, Air Force Institute of Technology, 1994.

Jia94. Xiaoping Jia. *ZTC: A Type Checker for Z.* Institute for Software Engineering, Department of Computer Science and Information Systems, 1994.

Lig91. David Lightfoot. *Formal Specification Using Z.* MacMillan Education Ltd, 1991.

NS91. Allan Norcliffe and Gil Slater. *Mathematics of Software Construction.* Ellis Horwood Limited, Market Cross House, Cooper Street, Chichester, England, 1991.

RBP+91. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall Inc., 1991.

Rea. Reasoning Systems Inc., 3260 Hillview Ave., Palo Alto, CA 94304. *Refine User's Guide.*

Sar94. Michael Sarchet. Modeling Workload Effectiveness and Efficiency of Air Force Wing Command and Control. Master's thesis, Air Force Institute of Technology, 1994.

Smi90. Douglas Smith. KIDS: A Semi-Automatic Program Development System. *IEEE Software Transactions*, 16(9):1024–1043, Sep 1990.

Sri91. Yellamraju V. Srinivas. Algebraic Specification. In Ruben Prieto-Diaz and Guillerma Arango, editors, *Domain Analysis and Software Systems Modeling*, pages 94–115. IEEE Software Press, September 1991.

Sys90. Reasoning Systems. *Dialect User's Guide.* Reasoning Systems, 1990.

USA93. USAF. Guidelines for Successful Acquisition and Management of Software Intensive Systems - Draft. AFPAM 63-115, 1993, November 1993.

Wab94. Kathleen Wabiszewksi. Unification of Larch and Z-Based Object Models To Support Algebraically-Based Design Refinement: The Z Perspective. Master's thesis, Air Force Institute of Technology, 1994.

Win87. Jeannette M. Wing. Writing Larch Interface Language Specifications. In *ACM Transactions on Programming Languages and Systems*, volume 9, pages 1–24, 1987.

Win91. Tim Winkler. *Programming in OBJ and Maude*, chapter 12, pages 229–277. Functional Programming, Concurrency, Simulation and Automated Reasoning. Springer-Verlag, 1991.

Vita

Captain Catherine J. Lin was born on May 26, 1966 in Chicago, Illinois and graduated from Freedom High School in Bethlehem, Pennsylvania in 1984. She was awarded a Bachelor's Degree in Electrical Engineering and Engineering and Public Policy at Carnegie Mellon University in Pittsburgh, Pennsylvania, in May 1988. She received her commission as a distinguished graduate through the Reserve Officer Training Program. As a lieutenant, she was assigned as the primary project engineer for the Software Technology for Adaptable, Reliable Systems program at the Systems and Software Design Center, Electronic Systems Center (ESC), Massachusetts. In August 1991, Captain Lin was selected to serve as an Air Force resident affiliate at the Software Engineering Institute (SEI). During this period, she fostered Air Force, Army, and Navy cooperation on the DoD Software Reuse Initiative. In addition, she established herself as a Software Capability Evaluation (SCE) expert by conducting over 12 SCEs on source selection and contract monitoring efforts. Upon returning from the SEI, Captain Lin was selected as the Deputy Program Manager for the Central Archive for Reusable Defense Software program. In May, 1993, she entered the Air Force Institute of Technology at Wright-Patterson AFB, Ohio, to pursue a Master of Science degree in Computer Science. Upon graduation, Captain Lin will be assigned to the Headquarters Cost Analysis Agency to provide the software expertise in their assessment efforts.

Permanent address: 7608 Lone Lane, Coopersburg, PA 18036

Appendix H. REFINE *Code for the* ULARCH *Domain Model*

The following REFINE specification defines the Larch Extensions to the Unified Domain Model. The original Larch domain model was revised to use the core unified domain model object classes and map attributes. The Larch extensions include Larch specific syntax that are specializations of a unified parent object class found in the Unified Domain Model.

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: ularch-dm.re

||#

% ------------------------------------------------------------------------
% Unified Domain Model Object Class Definitions
% ------------------------------------------------------------------------

var Unified-Object          : object-class subtype-of user-object

var DomainTheory            : object-class subtype-of Unified-Object

var DomainTheoryTypes       : object-class subtype-of Unified-Object
  var ObjectTheory          : object-class subtype-of DomainTheoryTypes
  var DynamicTheory         : object-class subtype-of DomainTheoryTypes
  var FunctionalTheory      : object-class subtype-of DomainTheoryTypes

var TheoryId                : object-class subtype-of Unified-Object
  var ObjectTheoryId        : object-class subtype-of TheoryId
  var DynamicTheoryId       : object-class subtype-of TheoryId
  var FunctionalTheoryId    : object-class subtype-of TheoryId

var TheoryBody              : object-class subtype-of Unified-Object
  var ObjectTheoryBody      : object-class subtype-of TheoryBody
  var DynamicTheoryBody     : object-class subtype-of TheoryBody
  var FunctionalTheoryBody  : object-class subtype-of TheoryBody

var TheoryDecl              : object-class subtype-of Unified-Object
  var ObjectTheoryDecl      : object-class subtype-of TheoryDecl
  var DynamicTheoryDecl     : object-class subtype-of TheoryDecl
  var FunctionalTheoryDecl  : object-class subtype-of TheoryDecl

var SignatureDecl           : object-class subtype-of Unified-Object
var ContextRef              : object-class subtype-of Unified-Object
```

```
var TheoryAxioms              : object-class subtype-of Unified-Object
  var ObjectTheoryAxioms      : object-class subtype-of TheoryAxioms
  var DynamicTheoryAxioms     : object-class subtype-of TheoryAxioms
  var FunctionalTheoryAxioms: object-class subtype-of TheoryAxioms

%-------------------------------------------------------------
% Larch Specific Class Objects
% The following object classes are specilizations of the Unified
% Domain model and refer to the specific syntax of the Larch
% Language.  Specific identifier objects have been defined
% in order to make the Refine program more understandable
% and more descriptive.
%-------------------------------------------------------------
var NameObj                   : object-class subtype-of Unified-Object
  var NameObj1                : object-class subtype-of NameObj
  var NameObj2                : object-class subtype-of NameObj
var OpForm                    : object-class subtype-of Unified-Object
  var OpForm-1                : object-class subtype-of OpForm
  var OpForm-2                : object-class subtype-of OpForm
  var OpForm-3                : object-class subtype-of OpForm
  var OpForm-4                : object-class subtype-of OpForm
  var OpForm-5                : object-class subtype-of OpForm
  var OpForm-6                : object-class subtype-of OpForm

var OpenSym                   : object-class subtype-of Unified-Object
  var OpenSym1                : object-class subtype-of OpenSym
  var OpenSym2                : object-class subtype-of OpenSym
  var OpenSym3                : object-class subtype-of OpenSym

var CloseSym                  : object-class subtype-of Unified-Object
  var CloseSym1               : object-class subtype-of CloseSym
  var CloseSym2               : object-class subtype-of CloseSym
  var CloseSym3               : object-class subtype-of CloseSym

var MarkerSym                 : object-class subtype-of Unified-Object

var ExternalRef               : object-class subtype-of ContextRef
  var Includes                : object-class subtype-of ExternalRef
  var Assumes                 : object-class subtype-of ExternalRef
var ShorthandRef              : object-class subtype-of ContextRef
  var Enumeration             : object-class subtype-of ShorthandRef
  var Tuple-Obj               : object-class subtype-of ShorthandRef
  var Union-Obj               : object-class subtype-of ShorthandRef

var OpDecl                    : object-class subtype-of Unified-Object

var Field                     : object-class subtype-of Unified-Object
var Trait-Ref                 : object-class subtype-of Unified-Object
  var Trait-Ref1              : object-class subtype-of Trait-Ref
  var Trait-Ref2              : object-class subtype-of Trait-Ref
var Renames                   : object-class subtype-of Unified-Object
```

```
  var Renames1          : object-class subtype-of Renames
  var Renames2          : object-class subtype-of Renames
var Replaces            : object-class subtype-of Unified-Object
var Parameter           : object-class subtype-of Unified-Object
var SortSet             : object-class subtype-of Unified-Object

var Op                  : object-class subtype-of Unified-Object
  var SimpleOp          : object-class subtype-of Op
    var UserOp          : object-class subtype-of SimpleOp
    var Multiply        : object-class subtype-of SimpleOp
    var Addition        : object-class subtype-of SimpleOp
    var Subtraction     : object-class subtype-of SimpleOp
    var Divide          : object-class subtype-of SimpleOp
    var LessThan        : object-class subtype-of SimpleOp
    var LessThanEq      : object-class subtype-of SimpleOp
    var Equality        : object-class subtype-of SimpleOp
    var GreaterThan     : object-class subtype-of SimpleOp
    var GreaterThanEq   : object-class subtype-of SimpleOp
    var EqualSym        : object-class subtype-of SimpleOp
    var NotEqual        : object-class subtype-of SimpleOp
  var LogicalOp         : object-class subtype-of Op
    var And-Obj         : object-class subtype-of LogicalOp
    var Or-Obj          : object-class subtype-of LogicalOp
    var Not-Obj         : object-class subtype-of LogicalOp
    var Implies         : object-class subtype-of LogicalOp

var Identifier          : object-class subtype-of Unified-Object
  var IdName            : object-class subtype-of Identifier
  var IntExpr           : object-class subtype-of Identifier
  var RealExpr          : object-class subtype-of Identifier

var ReservedId          : object-class subtype-of Unified-Object
  var LInverse          : object-class subtype-of ReservedId
  var Superstar         : object-class subtype-of ReservedId
  var Bottom            : object-class subtype-of ReservedId
  var Top               : object-class subtype-of ReservedId
  var Superplus         : object-class subtype-of ReservedId

var genPartition        : object-class subtype-of Unified-Object
  var Generators        : object-class subtype-of genPartition
  var Partitions        : object-class subtype-of genPartition
var EqPart              : object-class subtype-of Unified-Object
  var EqPart1           : object-class subtype-of EqPart
  var EqPart2           : object-class subtype-of EqPart
var Operator            : object-class subtype-of Unified-Object
var quantifier          : object-class subtype-of Unified-Object
var VarDcl              : object-class subtype-of Unified-Object

var equation            : object-class subtype-of Unified-Object
  var equivEqn          : object-class subtype-of equation
```

```
var quanteq              : object-class subtype-of Unified-Object

var term                 : object-class subtype-of Unified-Object
    var andTerm          : object-class subtype-of term
    var orTerm           : object-class subtype-of term
    var addTerm          : object-class subtype-of term
    var subtractTerm     : object-class subtype-of term
    var multiplyTerm     : object-class subtype-of term
    var divideTerm       : object-class subtype-of term
    var tupleTerm2       : object-class subtype-of term

    var bracketedTerm    : object-class subtype-of term
    var notTerm          : object-class subtype-of term
    var operationTerm    : object-class subtype-of term
      var operationTerm1   : object-class subtype-of operationTerm
      var operationTerm2   : object-class subtype-of operationTerm
    var specialTerm      : object-class subtype-of term
    var simpleTerm       : object-class subtype-of term
    var tupleTerm1       : object-class subtype-of term
    var ltTerm           : object-class subtype-of term
    var lteTerm          : object-class subtype-of term
    var gtTerm           : object-class subtype-of term
    var gteTerm          : object-class subtype-of term
    var eqTerm           : object-class subtype-of term
    var notEqTerm        : object-class subtype-of term
    var impliesTerm      : object-class subtype-of term
    var simpleOpTerm     : object-class subtype-of term
    var simpleOpTerm2    : object-class subtype-of term

  var ifStatement        : object-class subtype-of term
    var ifTerm           : object-class subtype-of ifStatement
    var elseTerm         : object-class subtype-of ifStatement
    var thenTerm         : object-class subtype-of ifStatement

var expression           : object-class subtype-of Unified-Object
  var bracketedExpr      : object-class subtype-of expression
  var primaryExpr        : object-class subtype-of expression
    var Expr1            : object-class subtype-of primaryExpr
    var Expr2            : object-class subtype-of primaryExpr
    var Expr3            : object-class subtype-of primaryExpr
    var Expr4            : object-class subtype-of primaryExpr
    var Expr5            : object-class subtype-of primaryExpr

var barmarker            : object-class subtype-of Unified-Object

var Consequences         : object-class subtype-of Unified-Object
  var Consequences1      : object-class subtype-of Consequences
  var Consequences2      : object-class subtype-of Consequences
  var Conversion         : object-class subtype-of Unified-Object
  var Exempting          : object-class subtype-of Unified-Object
var ConseqPart           : object-class subtype-of Unified-Object
```

```
  var ConseqPart1       : object-class subtype-of ConseqPart
  var ConseqPart2       : object-class subtype-of ConseqPart

%-------------------------------------------------------------------
% Unified Model Attribute Declarations for Branches in Tree Structure
%-------------------------------------------------------------------
var theory-types         : map(DomainTheory, set(DomainTheoryTypes))
                           computed-using theory-types(x) = {}
%var theory-types         : map(DomainTheory, seq(DomainTheoryTypes)) = {||}

var theory-id            : map(DomainTheoryTypes, IdName) = {||}
var ot-id                  : map(DomainTheoryTypes, IdName) = {||}
var dt-id                  : map(DomainTheoryTypes, IdName) = {||}
var ft-id                  : map(DomainTheoryTypes, IdName) = {||}

var theory-body          : map(DomainTheoryTypes, TheoryBody) = {||}
var ot-body                : map(DomainTheoryTypes, TheoryBody) = {||}
var dt-body                : map(DomainTheoryTypes, TheoryBody) = {||}
var ft-body                : map(DomainTheoryTypes, TheoryBody) = {||}

var theory-decl          : map(TheoryBody, TheoryDecl) = {||}
var context-refs         : map(TheoryDecl, set(ContextRef))
                           computed-using context-refs(x) = {}
var signature-decl       : map(TheoryDecl, SignatureDecl) = {||}

var theory-axioms        : map(TheoryBody, set(TheoryAxioms))
                           computed-using theory-axioms(x) = {}

%-------------------------------------------------------------
% Larch Attribute Declarations for Branches in Tree Structure
%-------------------------------------------------------------
var id                   : map(Unified-Object, IdName) = {||}
var trait-id             : map(Unified-Object, IdName) = {||}
var parameter-id         : map(Unified-Object, NameObj) = {||}
var sort-id              : map(Unified-Object, IdName) = {||}
var old-name-id          : map(Unified-Object, NameObj) = {||}
var new-name-id          : map(Unified-Object, NameObj) = {||}
var op-id                : map(Unified-Object, IdName) = {||}
var simple-id            : map(Unified-Object, Identifier) = {||}
var simple-id2           : map(Unified-Object, Identifier) = {||}
var element-ids          : map(Unified-Object, seq(IdName)) = {||}
var trait-ids            : map(Unified-Object, seq(IdName)) = {||}
var sort-ids             : map(Unified-Object, seq(IdName)) = {||}
var var-id               : map(Unified-Object, IdName) = {||}
var var-ids              : map(Unified-Object, seq(IdName)) = {||}
var field-id             : map(Unified-Object, IdName) = {||}
var field-ids            : map(Unified-Object, seq(IdName)) = {||}
var domain-ids           : map(Unified-Object, seq(IdName)) = {||}
var range-id             : map(Unified-Object, IdName) = {||}
var reserved-id          : map(Unified-Object, ReservedId) = {||}
```

```
var int                 : map(Unified-Object, integer) = {||}
var num-real            : map(Unified-Object, real) = {||}

var trait-parameter     : map(DomainTheoryTypes, seq(Parameter)) = {||}
var field-defs          : map(Unified-Object, seq(Field)) = {||}
var sort-def            : map(Unified-Object, SortSet) = {||}

var op-decls            : map(SignatureDecl, set(OpDecl))
                          computed-using op-decls(x) = {}

var traitref            : map(Unified-Object, seq(Trait-Ref)) = {||}
var renaming            : map(Trait-Ref, Renames) = {||}
var replacement         : map(Unified-Object, seq(Replaces)) = {||}

var any-op              : map(Unified-Object, Op) = {||}
var simple-op           : map(Unified-Object, SimpleOp) = {||}
var simple-ops          : map(Unified-Object, seq(SimpleOp)) = {||}
var logical-op          : map(Unified-Object, LogicalOp) = {||}
var lhs-marker          : map(Unified-Object, MarkerSym) = {||}
var rhs-marker          : map(Unified-Object, MarkerSym) = {||}
var if-marker           : map(Unified-Object, MarkerSym) = {||}
var then-marker         : map(Unified-Object, MarkerSym) = {||}
var else-marker         : map(Unified-Object, MarkerSym) = {||}
var markers             : map(Unified-Object, seq(MarkerSym)) = {||}
var open-sym            : map(Unified-Object, OpenSym) = {||}
var close-sym           : map(Unified-Object, CloseSym) = {||}
var name-id             : map(Unified-Object, NameObj) = {||}
var name-ids            : map(Unified-Object, seq(NameObj)) = {||}
var op-form             : map(NameObj, OpForm) = {||}

var gen-Partition       : map(Unified-Object, genPartition) = {||}
var gen-Partitions      : map(Unified-Object, seq(genPartition)) = {||}
var op-erator           : map(Unified-Object, Operator) = {||}
var op-erators          : map(Unified-Object, seq(Operator)) = {||}
var eq-part1            : map(TheoryAxioms, EqPart) = {||}
var eq-part2            : map(ConseqPart2, EqPart) = {||}
var qu-antifier         : map(Unified-Object, Quantifier) = {||}
var var-Dcl             : map(Unified-Object, VarDcl) = {||}
var var-Dcls            : map(Unified-Object, seq(VarDcl)) = {||}
var eq-uations1         : map(EqPart, set(equation))
                          computed-using eq-uations1(x) = {}
var eq-uations2         : map(ConseqPart1, set(equation))
                          computed-using eq-uations2(x) = {}

var lhs-term            : map(Unified-Object, term) = {||}
var rhs-term            : map(Unified-Object, term) = {||}
var te-rm               : map(Unified-Object, term) = {||}
var te-rms              : map(Unified-Object, seq(term)) = {||}
var bracketed-term      : map(Unified-Object, bracketedTerm) = {||}
var if-Term             : map(Unified-Object, term) = {||}
var then-Term           : map(Unified-Object, term) = {||}
```

```
var else-Term            : map(Unified-Object, term) = {||}

var args                 : map(Unified-Object, seq(expression)) = {||}
var arg-1                : map(Unified-Object, primaryExpr) = {||}
var arg-2                : map(Unified-Object, primaryExpr) = {||}
var arg-3                : map(Unified-Object, term) = {||}
var arg                  : map(Unified-Object, expression) = {||}
var bracketed-arg        : map(Unified-Object, bracketedExpr) = {||}

var quant-eq             : map(Unified-Object, seq(quanteq)) = {||}

var not-op               : map(Unified-Object, Not-Obj) = {||}
var neg-sign             : map(Unified-Object, Subtraction) = {||}

var con-sequences        : map(TheoryBody, Consequences) = {||}
var con-version          : map(Unified-Object, Conversion) = {||}
var ex-empting           : map(Unified-Object, Exempting) = {||}
var conseq-part          : map(Consequences, ConseqPart) = {||}

%-----------------------------------------------------------------
% Structure for Abstract Syntax Tree
%-----------------------------------------------------------------
form Define-Tree-Attributes-of-Unified-Specification
  Define-Tree-Attributes('DomainTheory, {'theory-types})&
  Define-Tree-Attributes('ObjectTheory, {'theory-id, 'trait-parameter, 'theory-body})&
  Define-Tree-Attributes('DynamicTheory, {'theory-id, 'trait-parameter, 'theory-body})&
  Define-Tree-Attributes('FunctionalTheory, {'theory-id, 'trait-parameter, 'theory-body})&
  Define-Tree-Attributes('TheoryBody, {'theory-decl, 'theory-axioms,
       'con-sequences})&
  Define-Tree-Attributes('TheoryDecl, {'context-refs, 'signature-decl})&
%  Define-Tree-Attributes('TheoryAxioms, {'gen-Partitions, 'qu-antifier, 'eq-uations})&
  Define-Tree-Attributes('TheoryAxioms, {'gen-Partitions, 'eq-part1})&

%% Larch Identifier Specializations
  Define-Tree-Attributes('Parameter, {'parameter-id, 'sort-def}) &
  Define-Tree-Attributes('NameObj1, {'op-id})&
  Define-Tree-Attributes('NameObj2, {'op-form})&
  Define-Tree-Attributes('OpForm-1, {'if-marker, 'then-marker, 'else-marker})&
  Define-Tree-Attributes('OpForm-2, {'lhs-marker, 'any-op, 'rhs-marker})&
  Define-Tree-Attributes('OpForm-3, {'lhs-marker, 'open-sym, 'markers,
     'close-sym, 'rhs-marker})&
  Define-Tree-Attributes('OpForm-4, {'lhs-marker, 'field-id})&
  Define-Tree-Attributes('OpForm-5, {'lhs-marker, 'rhs-marker, 'reserved-id})&
  Define-Tree-Attributes('OpForm-6, {'reserved-id})&
  Define-Tree-Attributes('SortSet, {'domain-ids, 'range-id})&

%% Larch Operation Declarations Specializations
  Define-Tree-Attributes('SignatureDecl, {'op-decls})&
  Define-Tree-Attributes('OpDecl, {'name-ids, 'sort-def})&

%% Larch Context References Specializations - External notations
```

```
  Define-Tree-Attributes('Includes, {'traitref})&
  Define-Tree-Attributes('Assumes, {'traitref})&

%% Larch Context References Specializations - Shorthand notations
  Define-Tree-Attributes('Tuple-Obj, {'sort-id, 'field-defs})&
  Define-Tree-Attributes('Union-Obj, {'sort-id, 'field-defs})&
  Define-Tree-Attributes('Enumeration, {'sort-id, 'element-ids})&

  Define-Tree-Attributes('Field, {'field-ids, 'sort-id})&
  Define-Tree-Attributes('Trait-Ref1, {'trait-id, 'renaming})&
  Define-Tree-Attributes('Trait-Ref2, {'trait-ids, 'renaming})&
  Define-Tree-Attributes('Renames1, {'replacement})&
  Define-Tree-Attributes('Renames2, {'name-ids, 'replacement})&
  Define-Tree-Attributes('Replaces, {'new-name-id, 'old-name-id, 'sort-def})&

%% Larch Generated Partitions Specializations
  Define-Tree-Attributes('Generators, {'sort-id, 'op-erators})&
  Define-Tree-Attributes('Partitions, {'sort-id, 'op-erators})&
  Define-Tree-Attributes('Operator, {'name-id, 'sort-def})&

%% Larch Axiomatic Equations Specializations
  Define-Tree-Attributes('EqPart1, {'eq-uations1})&
  Define-Tree-Attributes('EqPart2, {'qu-antifier, 'eq-uations1})&
  Define-Tree-Attributes('Quantifier, {'var-Dcls})&
  Define-Tree-Attributes('VarDcl, {'var-ids, 'sort-id})&
  Define-Tree-Attributes('equivEqn, {'lhs-term, 'rhs-term})&
  Define-Tree-Attributes('impliesTerm, {'arg-1, 'arg-2})&
  Define-Tree-Attributes('andTerm, {'arg-1, 'arg-2})&
  Define-Tree-Attributes('orTerm, {'arg-1, 'arg-2})&
  Define-Tree-Attributes('addTerm, {'arg-1, 'arg-2})&
  Define-Tree-Attributes('subtractTerm, {'arg-1, 'arg-2})&
  Define-Tree-Attributes('multiplyTerm, {'arg-1, 'arg-2})&
  Define-Tree-Attributes('divideTerm, {'arg-1, 'arg-2})&
  Define-Tree-Attributes('simpleTerm, {'arg-1, 'bracketed-arg})&
  Define-Tree-Attributes('specialTerm, {'arg-1, 'arg-2})&
  Define-Tree-Attributes('notTerm, {'not-op, 'lhs-term})&
  Define-Tree-Attributes('simpleOpTerm, {'simple-op, 'lhs-term})&
  Define-Tree-Attributes('bracketedTerm, {'bracketed-arg})&
  Define-Tree-Attributes('operationTerm, {'arg-1, 'op-id, 'arg-2})&
  Define-Tree-Attributes('ltTerm, {'arg-1, 'arg-2})&
  Define-Tree-Attributes('lteTerm, {'arg-1, 'arg-2})&
  Define-Tree-Attributes('gtTerm, {'arg-1, 'arg-2})&
  Define-Tree-Attributes('gteTerm, {'arg-1, 'arg-2})&
  Define-Tree-Attributes('eqTerm, {'arg-1, 'arg-2})&
  Define-Tree-Attributes('notEqTerm, {'arg-1, 'arg-2})&
  Define-Tree-Attributes('ifStatement, {'if-Term, 'then-Term, 'else-Term})&
  Define-Tree-Attributes('Expr1, {'te-rm, 'simple-id})&
  Define-Tree-Attributes('Expr2, {'simple-id})&
  Define-Tree-Attributes('Expr3, {'simple-id, 'args, 'simple-id2})&
  Define-Tree-Attributes('Expr4, {'sort-id, 'field-id})&
  Define-Tree-Attributes('bracketedExpr, {'open-sym, 'te-rms, 'close-sym, 'arg-2})&
```

```
%% Larch Consequences - checkable properties
  Define-Tree-Attributes('Consequences1, {'traitref, 'conseq-part, 'con-version})&
  Define-Tree-Attributes('ConseqPart1, {'gen-Partitions, 'eq-uations2})&
  Define-Tree-Attributes('ConseqPart2, {'gen-Partitions, 'eq-part2})&
  Define-Tree-Attributes('Conversion, {'op-erators, 'ex-empting})&
  Define-Tree-Attributes('Exempting, {'qu-antifier, 'te-rms})
```

Appendix I. REFINE *Code for the* ULARCH *Grammar*

The following REFINE specification defines the ULARCH Grammar. The grammar uses the unified domain model and the LARCH extensions (found in Appendices F and H) to construct its production rules. In addition to incorporating LaTeX specific notations, the ULARCH grammar uses the same common core objects and attributes as the UZ grammar to promote the notion of language inheritance. The production rules demonstrate DIALECT's use of object classes and attributes to form grammar rules. This approach is more intuitive simplifying a developer's task for forming tokens and creating a hierarchical structure for a formal language.

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: ularch-gram.re

||#

!!in-grammar('syntax)

grammar Ularch
   start-classes DomainTheory
   file-classes DomainTheory
   no-patterns
   comments "%" matching "
"
   case-sensitive
   symbol-continue-chars
   "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_="
   symbol-start-chars
   "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_="

  productions
   DomainTheory     ::=  ["\\documentstyle[fullpage,larch]{article}"
          "\\begin{document}" theory-types + "\\\\"
          "\\end{document}"] builds DomainTheory,

   ObjectTheory     ::=  ["\\begin{spec} %ObjectTheory" theory-id
          {[ "(" trait-parameter + ","  ")" ]} ":" "trait"
          theory-body
          "\\end{spec}"] builds ObjectTheory,

   DynamicTheory    ::=  ["\\begin{spec} %DynamicTheory" theory-id
          {[ "(" trait-parameter + ","  ")" ]} ":" "trait"
```

```
        theory-body
        "\\end{spec}"] builds DynamicTheory,

    FunctionalTheory  ::=  ["\\begin{spec} %FunctionalTheory" theory-id
        {[ "(" trait-parameter + ","  ")" ]} ":" "trait"
        theory-body
        "\\end{spec}"] builds FunctionalTheory,

    TheoryBody        ::=  [theory-decl theory-axioms * "" {con-sequences}]
                           builds TheoryBody,

%% Formal Parameters
    Parameter         ::=  [parameter-id {[":" sort-def]}] builds Parameter,
    SortSet           ::=  [domain-ids * "," "->" range-id] builds SortSet,
    NameObj1          ::=  [op-id] builds NameObj1,
    NameObj2          ::=  [op-form] builds NameObj2,
    OpForm-1          ::=  ["if" if-marker "then" then-marker
    "else" else-marker] builds OpForm-1,
    OpForm-2          ::=  [{lhs-marker} any-op {rhs-marker}] builds OpForm-2,
    OpForm-3          ::=  [{lhs-marker} open-sym markers * "," close-sym
    {rhs-marker}] builds OpForm-3,
    OpForm-4          ::=  [{lhs-marker} "." field-id] builds OpForm-4,

    OpenSym1          ::=  [ "[" ] builds OpenSym1,
    OpenSym2          ::=  [ "{" ] builds OpenSym2,
    OpenSym3          ::=  [ "\\<" ] builds OpenSym3,
    CloseSym1         ::=  [ "]" ] builds CloseSym1,
    CloseSym2         ::=  [ "}" ] builds CloseSym2,
    CloseSym3         ::=  [ "\\>" ] builds CloseSym3,
    MarkerSym         ::=  ["__"] builds MarkerSym,

%% Typed Expr Productions
    IdName            ::= [name] builds IdName,
    IntExpr           ::= [int] builds IntExpr,
    RealExpr          ::= [num-real] builds RealExpr,

%% Theory Declarations
     TheoryDecl       ::=  [context-refs * "" {signature-decl}] builds TheoryDecl,

%% trait body CONTEXTREFs
    %% external-refs
    Includes          ::=  ["includes" traitref + ","] builds Includes,
    Assumes           ::=  ["assumes"  traitref + ","] builds Assumes,
    %% shorthand-refs
    Tuple-Obj         ::=  [sort-id "tuple" "of" field-defs + ","] builds Tuple-Obj,
    Union-Obj         ::=  [sort-id "union" "of" field-defs + ","] builds Union-Obj,
    Enumeration       ::=  [sort-id "enumeration" "of" element-ids + ","] builds Enumeration,

    Field             ::=  [field-ids + "," ":" sort-id] builds Field,
    Trait-Ref1      ::=  [ trait-id {["(" renaming ")"]}] builds Trait-Ref1,
    Trait-Ref2      ::=  ["(" trait-ids + "," ")" {["(" renaming ")"]}] builds Trait-Ref2,
```

```
   Renames1       ::=  [replacement + "," { " " }] builds Renames1,
   Renames2       ::=  [name-ids + "," {["," replacement + ","]}]
                           builds Renames2,
   Replaces       ::=  [old-name-id "for" new-name-id {[":" sort-def]}]
                           builds Replaces,

%% Signature Declarations
   SignatureDecl      ::=  ["introduces" op-decls + ""] builds SignatureDecl,
   OpDecl             ::=  [name-ids + ","  ":" sort-def] builds OpDecl,


%% TheoryAxioms

   TheoryAxioms       ::=  ["asserts" gen-Partitions * "" eq-part1]
                           builds TheoryAxioms,
   Generators         ::=  [sort-id "generated by"
                           op-erators + ","] builds Generators,
   Partitions         ::=  [sort-id "partitioned by"
                            op-erators + ","] builds Partitions,
   Operator           ::=  [name-id {[":" sort-def]}] builds Operator,

   EqPart1            ::=  ["equations" eq-uations1 + ";"] builds EqPart1,
   EqPart2            ::=  [qu-antifier eq-uations1 + ";"] builds EqPart2,
   Quantifier         ::=  ["\\forall" var-Dcls + ","] builds Quantifier,
   VarDcl             ::=  [var-ids + "," ":" sort-id] builds VarDcl,

   equivEqn           ::=  [lhs-term {["==" rhs-term]}] builds equivEqn,

   notTerm            ::=  [not-op lhs-term] builds notTerm,
   simpleOpTerm       ::=  [simple-op {lhs-term}] builds simpleOpTerm,
   andTerm            ::=  [arg-1 "\\and" arg-2] builds andTerm,
   orTerm             ::=  [arg-1 "\\or" arg-2] builds orTerm,

   ifStatement        ::=  ["if" if-Term "then" then-Term "else"
   else-Term] builds ifStatement,

   impliesTerm        ::=  [arg-1 "\\implies" arg-2] builds impliesTerm,
%   impliesTerm        ::=  [arg-1 "\\implies" arg-3] builds impliesTerm,
% here arg-3 is mapped to a term.  This reduces one layer of () but
% increases the recursive layers making the tree deeper.
   addTerm            ::=  [arg-1 "+" arg-2] builds addTerm,
   subtractTerm       ::=  [arg-1 "-" arg-2] builds subtractTerm,
   multiplyTerm       ::=  [arg-1 "*" arg-2] builds multiplyTerm,
   divideTerm         ::=  [arg-1 "/" arg-2] builds divideTerm,
   operationTerm      ::=  [arg-1 "\\" op-id {arg-2}] builds operationTerm,
   specialTerm        ::=  [arg-1 "||" arg-2] builds specialTerm,

   simpleTerm         ::=  [arg-1 {bracketed-arg}] builds simpleTerm,
   bracketedTerm      ::=  [bracketed-arg] builds bracketedTerm,

   ltTerm             ::=  [arg-1 "<" arg-2] builds ltTerm,
```